36th International Conference on Concurrency Theory

CONCUR 2025, August 26–29, 2025, Aarhus, Denmark

Edited by

Patricia Bouyer

Jaco van de Pol

Editors

Patricia Bouyer
Université Paris-Saclay, CNRS, ENS Paris-Saclay, LMF, Gif-sur-Yvette, France
bouyer@lmf.cnrs.fr

Jaco van de Pol
Aarhus University, Denmark
jaco@cs.au.dk

ACM Classification 2012
Theory of Computation → Concurrency

ISBN 978-3-95977-389-8

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at https://www.dagstuhl.de/dagpub/978-3-95977-389-8.

Publication date
August, 2025

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at https://portal.dnb.de.

License
This work is licensed under a Creative Commons Attribution 4.0 International license (CC-BY 4.0): https://creativecommons.org/licenses/by/4.0/legalcode.
In brief, this license authorizes each and everybody to share (to copy, distribute and transmit) the work under the following conditions, without impairing or restricting the authors' moral rights:
- Attribution: The work must be attributed to its authors.

The copyright is retained by the corresponding authors.

Digital Object Identifier: 10.4230/LIPIcs.CONCUR.2025.0

ISBN 978-3-95977-389-8 **ISSN 1868-8969** **https://www.dagstuhl.de/lipics**

LIPIcs – Leibniz International Proceedings in Informatics

LIPIcs is a series of high-quality conference proceedings across all fields in informatics. LIPIcs volumes are published according to the principle of Open Access, i.e., they are available online and free of charge.

Editorial Board

- Christel Baier (TU Dresden, DE)
- Roberto Di Cosmo (Inria and Université Paris Cité, FR)
- Faith Ellen (University of Toronto, CA)
- Javier Esparza (TU München, DE)
- Holger Hermanns (Universität des Saarlandes, Saarbrücken, DE and Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Wadern, DE)
- Daniel Král' (Leipzig University, DE and Max Planck Institute for Mathematics in the Sciences, Leipzig, DE)
- Sławomir Lasota (University of Warsaw, PL)
- Meena Mahajan (Institute of Mathematical Sciences, Chennai, IN – Chair)
- Chih-Hao Luke Ong (Nanyang Technological University, SG)
- Eva Rotenberg (Technical University of Denmark, Lyngby, DK)
- Pierre Senellart (ENS, Université PSL, Paris, France)
- Alexandra Silva (Cornell University, Ithaca, US)

ISSN 1868-8969

https://www.dagstuhl.de/lipics

Contents

Preface
Patricia Bouyer and Jaco van de Pol 0:ix

Committees
..... 0:xi

Invited Talks

Linear Temporal Logic with Standpoint Modalities
Christel Baier 1:1–1:1

Towards Categorical Quantum Concurrency Theory
Chris Heunen 2:1–2:1

On-The-Fly Verification: Advancements in Dependency Graphs
Jiří Srba 3:1–3:5

Regular Papers

Monitorability for the Modal Mu-Calculus over Systems with Data: From Practice to Theory
Luca Aceto, Antonis Achilleos, Duncan Paul Attard, Léo Exibard, Adrian Francalanza, Anna Ingólfsdóttir, and Karoliina Lehtinen 4:1–4:21

Characterizations of Fragments of Temporal Logic over Mazurkiewicz Traces
Bharat Adsul, Paul Gastin, and Shantanu Kulkarni 5:1–5:20

Omega-Regular Verification and Control for Distributional Specifications in MDPs
S. Akshay, Ouldouz Neysari, and Đorđe Žikelić 6:1–6:19

Temporal Explorability Games
Pete Austin, Sougata Bose, Nicolas Mazzocchi, and Patrick Totzke 7:1–7:17

Model Checking as Program Verification by Abstract Interpretation
Paolo Baldan, Roberto Bruni, Francesco Ranzato, and Diletta Rigo 8:1–8:20

A Direct Reduction from Stochastic Parity Games to Simple Stochastic Games
Raphaël Berthon, Joost-Pieter Katoen, and Zihan Zhou 9:1–9:21

Abstract Subtyping for Asynchronous Multiparty Sessions
Laura Bocchi, Andy King, Maurizio Murgia, and Simon Thompson 10:1–10:19

A Sound and Complete Characterization of Fair Asynchronous Session Subtyping
Mario Bravetti, Luca Padovani, and Gianluigi Zavattaro 11:1–11:17

The Non-Cooperative Rational Synthesis Problem for SPEs and ω-Regular Objectives
Véronique Bruyère, Jean-François Raskin, Alexis Reynouard, and Marie Van Den Bogaard 12:1–12:23

36th International Conference on Concurrency Theory (CONCUR 2025).
Editors: Patricia Bouyer and Jaco van de Pol
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Languages of Boundedly-Ambiguous Vector Addition Systems with States
Wojciech Czerwiński and Łukasz Orlikowski 13:1–13:23

Reversible Pebble Transducers
Luc Dartois, Paul Gastin, Loïc Germerie Guizouarn, and Shankaranarayanan Krishna 14:1–14:22

On the Send-Synchronizability Problem for Mailbox Communication
Romain Delpy, Anca Muscholl, and Grégoire Sutre 15:1–15:20

Expectation in Stochastic Games with Prefix-Independent Objectives
Laurent Doyen, Pranshu Gaba, and Shibashis Guha 16:1–16:19

Just Verification of Mutual Exclusion Algorithms
Rob van Glabbeek, Bas Luttik, and Myrthe S. C. Spronck 17:1–17:25

A State-Based $O(m \log n)$ Partitioning Algorithm for Branching Bisimilarity
Jan Friso Groote and David N. Jansen 18:1–18:16

Time for Timed Monitorability
Thomas M. Grosen, Sean Kauffman, Kim G. Larsen, and Martin Zimmermann .. 19:1–19:20

Compositional Active Learning of Synchronizing Systems Through Automated Alphabet Refinement
Léo Henry, Mohammad Reza Mousavi, Thomas Neele, and Matteo Sammartino .. 20:1–20:22

Quantitative Language Automata
Thomas A. Henzinger, Pavol Kebis, Nicolas Mazzocchi, and N. Ege Saraç 21:1–21:24

Partial-Order Reduction Is Hard
Frédéric Herbreteau, Sarah Larroze-Jardiné, and Igor Walukiewicz 22:1–22:20

First-Order Store and Visibility in Name-Passing Calculi
Daniel Hirschkoff, Iwan Quémerais, and Davide Sangiorgi 23:1–23:21

Expressive Equivalence Between Decidable Freeze and Metric Timed Temporal Logics.
Hsi-Ming Ho, Shankara Narayanan Krishna, Khushraj Madnani, Rupak Majumdar, and Paritosh Pandya 24:1–24:24

On-The-Fly Symbolic Algorithm for Timed ATL with Abstractions
Nicolaj Ø. Jensen, Kim G. Larsen, Didier Lime, and Jiří Srba 25:1–25:19

Optimal Concolic Dynamic Partial Order Reduction
Mohammad Hossein Khoshechin Jorshari, Michalis Kokologiannakis, Rupak Majumdar, and Srinidhi Nagendra 26:1–26:22

Coverage Games
Orna Kupferman and Noam Shenwald 27:1–27:23

Arbitrary-Arity Tree Automata for QCTL
François Laroussinie and Nicolas Markey 28:1–28:20

Galois Energy Games: To Solve All Kinds of Quantitative Reachability Problems
Caroline Lemke and Benjamin Bisping 29:1–29:19

Open Bisimilarity for the π-Calculus with Mismatch
Tiange Liu, Alwen Tiu, and Ross Horne 30:1–30:20

Compositional Reasoning for Parametric Probabilistic Automata
Hannah Mertens, Tim Quatmann, and Joost-Pieter Katoen 31:1–31:20

Resolving Nondeterminism by Chance
Soumyajit Paul, David Purser, Sven Schewe, Qiyi Tang, Patrick Totzke, and Di-De Yen 32:1–32:22

Chance and Mass Interpretations of Probabilities in Markov Decision Processes
Yun Chen Tsai, Kittiphon Phalakarn, S. Akshay, and Ichiro Hasuo 33:1–33:19

New Fault Domains for Conformance Testing of Finite State Machines
Frits Vaandrager and Ivo Melse 34:1–34:22

From Bisimulation to Traces: The Impact of Parallel Composition on Finite Bases
Rowin Versteeg, Valentina Castiglioni, and Bas Luttik 35:1–35:18

Explainability is a Game for Probabilistic Bisimilarity Distances
Emily Vlasman, Anto Nanah Ji, James Worrell, and Franck van Breugel 36:1–36:20

Prophecies All the Way: Game-Based Model-Checking for HyperQPTL Beyond $\forall^*\exists^*$
Sarah Winter and Martin Zimmermann 37:1–37:18

Reachability in Vector Addition System with States Parameterized by Geometric Dimension
Yangluo Zheng 38:1–38:18

Denotational Semantics for Probabilistic and Concurrent Programs
Noam Zilberstein, Daniele Gorla, and Alexandra Silva 39:1–39:24

Preface

This volume contains the contributions accepted for the 36th International Conference on *Concurrency Theory* (CONCUR), held on 26-29 August, 2025. CONCUR serves as an annual scientific forum for researchers, developers, and students working to expand the field of concurrency theory and its applications. CONCUR 2025 was organized in Aarhus, Denmark as part of CONFEST 2025.

Along with CONCUR, CONFEST also featured the 30th International Conference on *Formal Methods for Industrial Critical Systems* (FMICS) and the combined International Conferences *Quantitative Evaluation of SysTems* and *Formal Modeling and Analysis of Timed Systems* (QEST+FORMATS), as well as the following six workshops: *Behavioural Metrics and Quantitative Logics* (BMQL), *Expressiveness in Concurrency and Structural Operational Semantics* (EXPRESS/SOS), *Formal Methods in Quantum Computing* (FMQC), *Recent Advances in Concurrency and Logic* (RADICAL), *Synthesis of Complex Parameters* (SYNCOP), and *Principles of Formal Quantitative Analysis* (PFQA), celebrating the birthday of Christel Baier.

CONCUR 2025 received 83 submissions and accepted 36 papers for presentation at the conference. We are grateful for the hard work of our program committee as well as the many external experts who produced 261 reviews in total, at least 3 per paper, and engaged in lively discussions. The quality of the submitted papers was very high, so there was a tough competition among the papers to be admitted to the final programme.

Besides the submitted scientific contributions, the program also contained a number of invited talks. We would like to thank the CONCUR invited speakers:

- Christel Baier (Technical University Dresden, Germany)
- Chris Heunen (University of Edinburgh, UK)
- Jiri Srba (Aalborg University, Denmark)

The proceedings of CONCUR 2025 are freely available through the LIPIcs series. We thank Andreas Pavlogiannis, general CONFEST co-chair, as well as the rest of the organizing team, and Aarhus University for their assistance in organizing CONFEST 2025.

Patricia Bouyer and Jaco van de Pol

36th International Conference on Concurrency Theory (CONCUR 2025).
Editors: Patricia Bouyer and Jaco van de Pol
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Committees

Program Committee

- Luca Aceto, Reykjavik University, Iceland & Gran Sasso Science Institute, Italy
- Guy Avni, University of Haifa, Israel
- Ezio Bartocci, TU Wien, Austria
- Michael Blondin, Université de Sherbrooke, Canada
- Patricia Bouyer, CNRS, France (chair)
- Janna Burman, Université Paris-Saclay, France
- Michaël Cadilhac, DePaul University, USA
- Pedro R. D'Argenio, Universidad Nacional de Córdoba, Argentina
- Josee Desharnais, Laval University, Canada
- Pierre Ganty, IMDEA Software Institute, Spain
- Rob van Glabbeek, University of Edinburgh, UK
- Christoph Haase, University of Oxford, UK
- Tingting Han, Birkbeck, University of London, UK
- Sebastian Junges, Radboud University, The Netherlands
- Jan Kretinsky, Masaryk University, Czech Republic
- S Krishna, IIT Bombay, India
- Sławomir Lasota, University of Warsaw, Poland
- Rupak Majumdar, Max Planck Institute for Software Systems, Germany
- Paul-André Melliès, CNRS, Université de Paris Cité, France
- Benjamin Monmege, Aix-Marseille Université, France
- Wojciech Penczek, Polish Academy of Sciences, Poland
- Nir Piterman, University of Gothenburg, Sweden
- Jaco van de Pol, Aarhus University, Denmark (chair)
- Mickael Randour, F.R.S.-FNRS & Université de Mons, Belgium
- Ana Sokolova, University of Salzburg, Austria
- B Srivathsan, Chennai Mathematical Institute, India
- Marielle Stoelinga, University of Twente, The Netherlands
- Bow-Yaw Wang, Academia Sinica, Taiwan
- Dominik Wojtczak, University of Liverpool, UK
- Zhilin Wu, Chinese Academy of Sciences, China
- Mingsheng Ying, University of Technology Sydney, Australia
- Martin Zimmermann, Aalborg University, Denmark

Steering Committee

- Luca Aceto, Reykjavik University, Iceland, and Gran Sasso Science Institute, Italy
- Christel Baier, Technical University Dresden, Germany
- Pedro R. D'Argenio, Universidad Nacional de Córdoba, Argentina
- Wan Fokkink, Vrije Universiteit Amsterdam, The Netherlands (chair)
- Catuscia Palamidessi, Inria, France
- Jiri Srba, Aalborg University, Denmark

36th International Conference on Concurrency Theory (CONCUR 2025).
Editors: Patricia Bouyer and Jaco van de Pol
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Linear Temporal Logic with Standpoint Modalities

Christel Baier
Technische Universität Dresden, Germany

Abstract

Standpoint logics have been introduced in recent work by Alvarez et al. as a multi-modal logic to reason about the integrated knowledge of multiple agents that might have different, possibly contradicting views ("standpoints"). The essential new feature are modalities for expressing that a property is conceivable according to the view of an agent. The talk considers the model checking problem for a standpoint extension of classical linear temporal logic (LTL) with five semantics for the standpoint modalities. The semantics differ in the information an agent can extract from the history. Starting with a generic non-elementary model checking algorithm that is applicable to all five semantics, a more detailed complexity analysis leads to improved upper bounds for four of the semantics. In three cases, the model checking problem turns out to be PSPACE-complete, i.e., not harder than the model checking problem for classical LTL, which stands in contrast to the known EXPSPACE-completeness result for the satisfiability problem for standpoint LTL.

2012 ACM Subject Classification Theory of computation → Concurrency; Theory of computation → Modal and temporal logics

Keywords and phrases Linear Temporal Logic, LTL, standpoint logics, multi-modal logic, model checking

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2025.1

Category Invited Talk

© Christel Baier;
licensed under Creative Commons License CC-BY 4.0
36th International Conference on Concurrency Theory (CONCUR 2025).
Editors: Patricia Bouyer and Jaco van de Pol; Article No. 1; pp. 1:1–1:1

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Towards Categorical Quantum Concurrency Theory

Chris Heunen

University of Edinburgh, UK

Abstract

Quantum computing inherently has concurrent aspects. Even with only local operations, qubits can influence each other. This ability leads to genuinely new quantum communication protocols, but also raises even thornier questions of causality than in classical concurrent computing. Monoidal categories and their string diagrams form a convenient and popular language for quantum computing. After an introduction to quantum concurrency, I will discuss the framework of tensor topology, which aims to analyse the interaction of several agents in monoidal categories, using notions from sheaf theory and ordered locales.

2012 ACM Subject Classification Theory of computation → Quantum computation theory; Theory of computation → Categorical semantics; Theory of computation → Concurrency

Keywords and phrases Quantum computing, causality, monoidal categories, tensor topology

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2025.2

Category Invited Talk

References

1 C. Constantin, N. Dicaire, and C. Heunen. Localisable monads. In *Computer Science Logic*, volume 15 of *Leibniz International Proceedings in Informatics*, pages 1–17, 2022. doi:10.4230/LIPIcs.CSL.2022.15.

2 P. Enrique Moliner, C. Heunen, and S. Tull. Space in monoidal categories. In *Quantum Physics and Logic*, volume 266 of *Electronic Proceedings in Theoretical Computer Science*, pages 399–410, 2017. doi:10.4204/EPTCS.266.25.

3 P. Enrique Moliner, C. Heunen, and S. Tull. Tensor topology. *Journal of Pure and Applied Algebra*, 224(10):106378, 2020. doi:10.1016/j.jpaa.2020.106378.

4 C. Heunen and J. S. Pacaud Lemay. Tensor-restriction categories. *Theory and Applications of Categories*, 37(21):635–670, 2021.

5 C. Heunen and N. van der Schaaf. Ordered locales. *Journal of Pure and Applied Algebra*, 228(7):107654, 2024. doi:10.1016/j.jpaa.2024.107654.

6 C. Heunen and J. Vicary. *Categories for Quantum Theory*. Oxford University Press, 2019. doi:10.1093/oso/9780198739623.001.0001.

7 R. Soares Barbosa and C. Heunen. Sheaf representation of monoidal categories. *Advances in Mathematics*, 416:108900, 2023. doi:10.1016/j.aim.2023.108900.

© Chris Heunen;
licensed under Creative Commons License CC-BY 4.0

On-The-Fly Verification: Advancements in Dependency Graphs

Jiří Srba
Department of Computer Science, Aalborg University, Denmark

Abstract

Dependency graphs have emerged as a versatile and powerful formalism with wide-ranging applications in formal verification. In this extended abstract, we provide an overview of selected advancements in on-the-fly verification techniques based on dependency graphs, focusing on the recent developments, optimizations and generalizations of this generic verification framework.

2012 ACM Subject Classification Theory of computation

Keywords and phrases dependency graphs, Boolean equation systems, on-the-fly algorithms, fixed-point computation, applications

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2025.3

Category Invited Talk

Acknowledgements I want to thank many of my co-authors for their contributions to the development of the dependency graph framework and its applications, including J.R. Andersen, N. Andersen, A.E. Dalsgaard, S. Enevoldsen, P. Fogh, M.M. Hansen, L.S. Jensen, M.C. Jensen, N.O. Jensen, P.G. Jensen, T.S. Jepsen, I. Kaufmann, K.G. Larsen, D. Lime, A. Mariegaard, S.M. Nielsen, S.R. Olesen, M.Chr. Olesen and S. Pastva.

1 Introduction

On-the-fly model checking (also called local model checking) is an efficient approach to formal verification that explores complex systems incrementally, enabling analysis of large models without necessarily constructing their entire state-space upfront [28, 29, 33, 5, 3, 34].

Dependency graphs, introduced by Liu and Smolka [31], are directed graphs where nodes are associated with Boolean values and hyperedges connect a source node to a number of target nodes in order to represent causal dependencies in the graph. An assignment of nodes to Boolean values 0 (false) and 1 (true) is a fixed-point assignment if for every hyperedge in the graph, the source node must be assigned the value 1 whenever all of the target nodes already have the value 1. Clearly, assigning 1 to all nodes gives us a fixed-point assignment, however, we are interested in the minimum fixed-point assignment which can be obtained by initially assigning 0 to all nodes and repeatedly improving the value 0 to 1 as long as this is required by some hyperedge. Note that we allow hyperedges with an empty set of target nodes which implies that the source node must necessarily obtain the value 1 in the minimum fixed-point assignment.

We are usually only interested in the minimum fixed-point value of a specific root node, which allows us in certain situations to explore only a subset of nodes in order to determine the value of the root node. In their seminal work [31], Liu and Smolka provided an elegant, linear-time algorithm for computing the minimum fixed-point assigment of the root node in a given dependency graph where the nodes of the graph are constructed on-the-fly during a forward exploration and where the newly discovered assignments of value 1 are back-propagated along the hyperedges that are stored in a dynamically updated dependency list. Similar ideas of local algorithms were developed also for a related formalism of Boolean Equation Systems (BES) [1, 2, 30].

© Jiří Srba;
licensed under Creative Commons License CC-BY 4.0

The generic formalism of a dependency graph, together with the efficient linear-time algorithm for computing fixed points, has found numerous applications in the equivalence and model checking community. An overview of such applications can be found in [14, 16]. Among others, dependency graphs stand behind the verification engine UPPAAL TIGA for strategy synthesis in timed games [7] and are the key-stone of the CTL verification engine for Petri nets [10, 23], implemented in the tool TAPAAL [12]. The tool CAAL [4] is completely built on the dependency graph framework, supporting both bisimulation and model checking of CCS processes, including the possibility of playing games based on the generated minimum fixed-point assignment.

The dependency graph formalism was further extended towards more expressive domains for node assignments like integer domains for weighted CTL [18, 19] and even more complex domains used for PCTL model checking [32], further generalized to weighted PCTL [21] and multiplayer stochastic games [13]. Dependency graphs showed their usefulness for the synthesis problems on multi-weighted games with branching conditions [20, 27] as well as timed-arc Petri net games with discrete time [24, 25]. Recently, dependency graphs were used for the verification and synthesis of (untimed) alternating-time logic (ATL) properties in concurrent games with multiple players [6] as well as for the timed variant of ATL that allows us to specify player-coalitions [22]. Parametric systems were also analyzed by means of dependency graphs [8].

Concurrently, there has been an on-going effort to further optimize and speed-up the algorithms for computing minimum fixed-point assignments on dependency graphs and their extensions. The certain-zero technique [10] introduces a back-propagation of the Boolean value 0 along the dependent hyperedges, as soon as it becomes clear that this value cannot be further improved to 1. This often allows us to achieve an early termination of the algorithms also for the negative cases where the root node can never obtain the value 1. Negation edges, introduced in [10], enable an on-the-fly computation of mixed minimum and maximum fixed-points, as needed for example for the encoding of logical negation. The work in [15] introduces the "ignore" function that can speed-up the evaluation of nodes by determining a subset of target nodes that are still relevant for the evaluation of the source node. This allows us to prune the list of dependencies of nodes and the paper [15] discusses a recursive algorithm for implementing such a pruning approach. A more light-weight but practically beneficial variant of the pruning approach, the so-called "elimination of detached regions" in the dependency graph [26], shows promising experimental results. Recently, the technique of inclusion checking that formalizes yet another version of on-the-fly pruning of the state-space has been defined in the dependency graph framework [22], further improving the performance of the algorithms. Furthermore, there exists a line of work that describes a successful distribution of the exploration work among a number of distributed processes (workers) that communicate via message passing [11, 10, 9].

As many of the applications mentioned above include the implementation of a particular variant of the minimum fixed-point algorithm for some given abstract domain, the work on abstract dependency graphs [15] generalizes all these single-purpose implementations into a unified framework and provides a single, efficient implementation[1] that can be instantiated to many different cases as outlined in [17]. If the node assignment is taking values from the abstract algebraic structure of a Noetherian partial order with the least element, the termination of the generic fixed-point computation algorithm is guaranteed.

[1] `https://launchpad.net/adg-tool/`

References

1 Henrik Reif Andersen. Model checking and Boolean graphs. In Bernd Krieg-Brückner, editor, *ESOP '92, 4th European Symposium on Programming, Rennes, France, February 26-28, 1992, Proceedings*, volume 582 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 1992. doi:10.1007/3-540-55253-7_1.

2 Henrik Reif Andersen. Model checking and Boolean graphs. *Theoretical Computer Science*, 126(1):3–30, 1994. doi:10.1016/0304-3975(94)90266-6.

3 Henrik Reif Andersen and Glynn Winskel. Compositional checking of satisfaction. In Kim Guldstrand Larsen and Arne Skou, editors, *Computer Aided Verification, 3rd International Workshop, CAV '91, Aalborg, Denmark, July, 1-4, 1991, Proceedings*, volume 575 of *Lecture Notes in Computer Science*, pages 24–36. Springer, 1991. doi:10.1007/3-540-55179-4_4.

4 J.R. Andersen, N. Andersen, S. Enevoldsen, M.M. Hansen, K.G. Larsen, S.R. Olesen, J. Srba, and J.K. Wortmann. CAAL: Concurrency workbench, Aalborg edition. In *Proceedings of the 12th International Colloquium on Theoretical Aspects of Computing (ICTAC'15)*, volume 9399 of *LNCS*, pages 573–582. Springer, 2015. doi:10.1007/978-3-319-25150-9_33.

5 Julian C. Bradfield and Colin Stirling. Local model checking for infinite state spaces. *Theor. Comput. Sci.*, 96(1):157–174, 1992. doi:10.1016/0304-3975(92)90183-G.

6 Falke B. Ø. Carlsen, Lars Bo P. Frydenskov, Nicolaj Ø. Jensen, Jener Rasmussen, Mathias M. Sørensen, Asger G. Weirsøe, Mathias C. Jensen, and Kim G. Larsen. CGAAL: Distributed on-the-fly ATL model checker with heuristics. *Electronic Proceedings in Theoretical Computer Science*, 390:99–114, September 2023. doi:10.4204/eptcs.390.7.

7 Franck Cassez, Alexandre David, Emmanuel Fleury, Kim G. Larsen, and Didier Lime. Efficient on-the-fly algorithms for the analysis of timed games. In Martín Abadi and Luca de Alfaro, editors, *CONCUR 2005 – Concurrency Theory*, volume 3653 of *LNCS*, pages 66–80, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. doi:10.1007/11539452_9.

8 Peter Christoffersen, Mikkel Hansen, Anders Mariegaard, Julian Trier Ringsmose, Kim Guldstrand Larsen, and Radu Mardare. Parametric Verification of Weighted Systems. In Étienne André and Goran Frehse, editors, *2nd International Workshop on Synthesis of Complex Parameters (SynCoP'15)*, volume 44 of *OpenAccess Series in Informatics (OASIcs)*, pages 77–90, Dagstuhl, Germany, 2015. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/OASIcs.SynCoP.2015.77.

9 A.E. Dalsgaard, S. Enevoldsen, P. Fogh, L.S. Jensen, P.G. Jensen, T.S. Jepsen, I. Kaufmann, K.G. Larsen, S.M. Nielsen, M.Chr. Olesen, S. Pastva, and J. Srba. A distributed fixed-point algorithm for extended dependency graphs. *Fundamenta Informaticae*, 161(4):351–381, 2018. doi:10.3233/FI-2018-1707.

10 A.E. Dalsgaard, S. Enevoldsen, P. Fogh, L.S. Jensen, T.S. Jepsen, I. Kaufmann, K.G. Larsen, S.M. Nielsen, M.Chr. Olesen, S. Pastva, and J. Srba. Extended dependency graphs and efficient distributed fixed-point computation. In *Proceedings of the 38th International Conference on Application and Theory of Petri Nets and Concurrency (Petri Nets'17)*, volume 10258 of *LNCS*, pages 139–158. Springer-Verlag, 2017. doi:10.1007/978-3-319-57861-3_10.

11 A.E. Dalsgaard, S. Enevoldsen, K.G. Larsen, and J. Srba. Distributed computation of fixed points on dependency graphs. In *Proceedings of Symposium on Dependable Software Engineering: Theories, Tools and Applications (SETTA'16)*, volume 9984 of *LNCS*, pages 197–212. Springer, 2016. doi:10.1007/978-3-319-47677-3_13.

12 A. David, L. Jacobsen, M. Jacobsen, K.Y. Jørgensen, M.H. Møller, and J. Srba. TAPAAL 2.0: Integrated development environment for timed-arc Petri nets. In *Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'12)*, volume 7214 of *LNCS*, pages 492–497. Springer-Verlag, 2012. doi:10.1007/978-3-642-28756-5_36.

13 S. Enevoldsen, M.C. Jensen, K.G. Larsen, A. Mariegaard, and J. Srba. Verification of multiplayer stochastic games via abstract dependency graphs. In *Proceedings of the 30th*

International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'20), volume 12561 of *LNCS*, pages 249–268. Springer, 2020. doi:10.1007/978-3-030-68446-4_13.

14 S. Enevoldsen, K.G. Larsen, A. Mariegaard, and J. Srba. Dependency graphs with applications to verification. *International Journal on Software Tools for Technology Transfer (STTT)*, 22:635–654, 2020. doi:10.1007/s10009-020-00578-9.

15 S. Enevoldsen, K.G. Larsen, and J. Srba. Abstract dependency graphs and their application to model checking. In *Proceedings of the 25th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'19)*, volume 11427 of *LNCS*, pages 316–333. Springer-Verlag, 2019. doi:10.1007/978-3-030-17462-0_18.

16 S. Enevoldsen, K.G. Larsen, and J. Srba. Model verification through dependency graphs. In *Proceedings of the 26th International SPIN Symposium on Model Checking of Software (SPIN'19)*, volume 11636 of *LNCS*, pages 1–19. Springer, 2019. doi:10.1007/978-3-030-30923-7_1.

17 S. Enevoldsen, K.G. Larsen, and J. Srba. Extended abstract dependency graphs. *International Journal on Software Tools for Technology Transfer (STTT)*, 24:49–65, 2022. doi:10.1007/s10009-021-00638-8.

18 J.F. Jensen, K.G. Larsen, J. Srba, and L.K. Oestergaard. Local model checking of weighted CTL with upper-bound constraints. In *Proceedings of International SPIN Symposium on Model Checking of Software (SPIN'13)*, volume 7976 of *LNCS*, pages 178–195. Springer-Verlag, 2013. doi:10.1007/978-3-642-39176-7_12.

19 J.F. Jensen, K.G. Larsen, J. Srba, and L.K. Oestergaard. Efficient model checking of weighted CTL with upper-bound constraints. *International Journal on Software Tools for Technology Transfer (STTT)*, 18(4):409–426, 2016. doi:10.1007/s10009-014-0359-5.

20 J.S. Jensen, I. Kaufmann, K.G. Larsen, S.M. Nielsen, and J. Srba. Model checking and synthesis for branching multi-weighted logics. *Journal of Logical and Algebraic Methods in Programming*, 105(1):28–46, 2019. doi:10.1016/j.jlamp.2019.02.001.

21 Mathias Claus Jensen, Anders Mariegaard, and Kim Guldstrand Larsen. Symbolic model checking of weighted PCTL using dependency graphs. In Julia M. Badger and Kristin Yvonne Rozier, editors, *NASA Formal Methods - 11th International Symposium, NFM 2019, Houston, TX, USA, May 7-9, 2019, Proceedings*, volume 11460 of *Lecture Notes in Computer Science*, pages 298–315. Springer, 2019. doi:10.1007/978-3-030-20652-9_20.

22 Nicolaj Ø. Jensen, Kim G. Larsen, Didier Lime, and Jiří Srba. On-the-fly symbolic algorithm for timed ATL with abstractions. In *36th International Conference on Concurrency Theory (CONCUR'25)*, pages 1–18. Schloss Dagstuhl, 2025. To appear.

23 N.O. Jensen, K.G. Larsen, and J. Srba. Token elimination in model checking of Petri nets. In *Proceedings of the 31st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'25)*, volume 15696 of *LNCS*, pages 211–230. Springer-Verlag, 2025. doi:10.1007/978-3-031-90643-5_11.

24 P.G. Jensen, K.G. Larsen, and J. Srba. Real-time strategy synthesis for timed-arc Petri net games via discretization. In *Proceedings of the 23rd International SPIN Symposium on Model Checking of Software (SPIN'16)*, volume 9641 of *LNCS*, pages 129–146. Springer-Verlag, 2016. doi:10.1007/978-3-319-32582-8_9.

25 P.G. Jensen, K.G. Larsen, and J. Srba. Discrete and continuous strategies for timed-arc Petri net games. *International Journal on Software Tools for Technology Transfer (STTT)*, 20(5):529–546, 2018. doi:10.1007/s10009-017-0473-2.

26 P.G. Jensen, K.G. Larsen, J. Srba, and N.J. Ulrik. Elimination of detached regions in dependency graph verification. In *Proceedings of the 29th International SPIN Symposium on Model Checking of Software (SPIN'23)*, volume 13872 of *LNCS*, pages 163–179. Springer-Verlag, 2023. doi:10.1007/978-3-031-32157-3_9.

27 I. Kaufmann, K.G. Larsen, and J. Srba. Synthesis for multi-weighted games with branching-time winning conditions. In *Proceedings of the 41st International Conference on Application and Theory of Petri Nets and Concurrency (Petri Nets'20)*, volume 12152 of *LNCS*, pages 46–66. Springer-Verlag, 2020. doi:10.1007/978-3-030-51831-8_3.

28 Kim Guldstrand Larsen. Proof system for hennessy-milner logic with recursion. In Max Dauchet and Maurice Nivat, editors, *CAAP '88, 13th Colloquium on Trees in Algebra and Programming, Nancy, France, March 21-24, 1988, Proceedings*, volume 299 of *Lecture Notes in Computer Science*, pages 215–230. Springer, 1988. doi:10.1007/BFb0026106.

29 Kim Guldstrand Larsen. Proof systems for satisfiability in Hennessy-Milner logic with recursion. *Theor. Comput. Sci.*, 72(2&3):265–288, 1990. doi:10.1016/0304-3975(90)90038-J.

30 Xinxin Liu, C. R. Ramakrishnan, and Scott A. Smolka. Fully local and efficient evaluation of alternating fixed points. In *Proceedings of TACAS'98*, volume 1384 of *LNCS*, pages 5–19. Springer, 1998. doi:10.1007/BFb0054161.

31 Xinxin Liu and Scott A. Smolka. Simple linear-time algorithms for minimal fixed points (extended abstract). In *Proceedings of ICALP'98*, volume 1443 of *LNCS*, pages 53–66, London, UK, UK, 1998. Springer-Verlag. doi:10.1007/BFB0055040.

32 Anders Mariegaard and Kim Guldstrand Larsen. Symbolic dependency graphs for PCTL model-checking. In Alessandro Abate and Gilles Geeraerts, editors, *15th Internation Conference on Formal Modeling and Analysis of Timed Systems (FORMATS'17)*, volume 10419 of *LNCS*, pages 153–169. Springer, 2017. doi:10.1007/978-3-319-65765-3_9.

33 Colin Stirling and David Walker. Local model checking in the modal mu-calculus. *Theor. Comput. Sci.*, 89(1):161–177, 1991. doi:10.1016/0304-3975(90)90110-4.

34 Glynn Winskel. A note on model checking the modal nu-calculus. *Theor. Comput. Sci.*, 83(1):157–167, 1991. doi:10.1016/0304-3975(91)90043-2.

Monitorability for the Modal Mu-Calculus over Systems with Data: From Practice to Theory

Luca Aceto
Dept. of Computer Science,
Reykjavik University, Iceland
Gran Sasso Science Institute, L'Aquila, Italy

Antonis Achilleos
Dept. of Computer Science,
Reykjavik University, Iceland

Duncan Paul Attard
University of Malta, Msida, Malta

Léo Exibard
LIGM, CNRS, Univ Gustave Eiffel,
F77454 Marne-la-Vallée, France

Adrian Francalanza
University of Malta, Msida, Malta

Anna Ingólfsdóttir
Dept. of Computer Science,
Reykjavik University, Iceland

Karoliina Lehtinen
CNRS, Aix-Marseille University, LIS,
Marseille, France

Abstract

Runtime verification consists in checking whether a system satisfies a given specification by observing the execution trace it produces. In the regular setting, the modal μ-calculus provides a versatile formalism for expressing specifications of the control flow of the system. This paper focuses on the *data* flow and studies an extension of that logic that allows it to express data-dependent properties, identifying fragments that can be verified at runtime and with what correctness guarantees. The logic studied here is closely related with register automata with *guessing*. That correspondence yields a monitor synthesis algorithm, and a strict hierarchy among the various fragments of the logic, in contrast to the regular setting. We then exhibit a fragment of the logic that can express all monitorable formulae in the logic without greatest fixed-points but not in the full logic, and show this is the best we can get.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Modal and temporal logics; Theory of computation → Automata over infinite objects

Keywords and phrases Runtime verification, monitorability, μHML with data, register automata

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2025.4

Related Version *Full Version*: https://arxiv.org/abs/2506.06172

1 Introduction

Runtime verification is an increasingly important lightweight validation technique that consists in checking a specification by observing an execution trace at runtime [14]. Not all system properties can be verified this way, e.g. those that mention behaviours that are not observed in the given trace, or limit behaviours such as "every request is always eventually granted". However, it can check properties for which an exhaustive state-space exploration is impractical, and verify systems whose model is unavailable, e.g. closed source code.

In the classical setting, system properties are typically expressed through formalisms whose models are (ω-)words or (ω-)trees, e.g. linear-time temporal logic (LTL), computation tree logic (CTL/CTL*) or the modal μ-calculus (equivalently, Hennessy-Milner logic with recursion), all falling within the realm of (ω-)regular behaviours [32]. While this setting enjoys numerous desirable properties (reasonable computational complexity, closure properties,

© Luca Aceto, Antonis Achilleos, Duncan Paul Attard, Léo Exibard, Adrian Francalanza, Anna Ingólfsdóttir, and Karoliina Lehtinen;
licensed under Creative Commons License CC-BY 4.0

correspondence with automata models, etc.), it falls short of capturing properties of the *data flow* of the system – what information it manipulates and how – since the alphabet of the traces or computation trees is assumed to be finite and typically small, corresponding to a focus on the *control* flow of the system – which signals it emits and when. The data flow has typically higher complexity, due to its unbounded nature, making it seem out of reach. However, due to the ubiquity of data manipulation and the increasing availability of computational power, numerous formal methods have shifted the focus to data, in runtime verification [41, 43], model-checking [20] and reactive synthesis [31, 33].

In the field of runtime verification, tools supporting monitoring of data-dependent properties of systems have been available for some time and have been applied in a variety of settings [2, 8, 11–13, 15, 16, 23, 28, 40, 47] (see also the surveys [34, 43]). However, to our mind, the systematic development of their theoretical underpinnings has lagged behind their practice. To quote Milner in [48] "the design of computing systems can only properly succeed if it is well grounded in theory". That quote motivates us to study the theoretical foundations of runtime monitoring for properties of data-dependent systems and to provide a systematic analysis of which properties can be monitored at runtime and with what correctness guarantees, paralleling our analysis in the regular setting [4].

To represent systems with data, we use data words and trees, whose elements are pairs of a letter from a finite alphabet and a value from an infinite domain, structured by a set of predicates to compare data values. In [41], the authors introduce a modal μ-calculus based formalism to express data properties that can be monitored at runtime. In [2], we introduced a variant with only the equality predicate. We provide an in-depth study of this logic by studying its expressiveness and monitorability. This reveals an intricate landscape where, in contrast to the ω-regular setting, most variations on the notion of monitor are *not* equivalent, demonstrating the need for distinct monitor models dependent on the property being monitored for. We also uncover a mistake in [7, Theorem 18], since what was believed to be a normal form is actually less expressive than the full monitor model. This observation may trigger development in the corresponding tool. The main contributions of the paper are:

- A formulation of the Hennessy-Milner logic with recursion over data words (Sec. 2) which refines the one in [2]. It is inspired from [41], with the equality predicate only (and hence without functions).
- The delineation of the HML^d fragment, capturing all completely monitorable properties expressible over data domains with only equality (Sec. 3.2 and Theorem 14).
- The delineation of the cHML^d fragment, which we show is monitorable for satisfactions by a natural extension of the monitor model in [4], along with its compositional synthesis algorithm (Sec. 3.3 and Theorem 17). This monitor model is moreover as expressive as alternating register automata with existential guessing [35, 36] (Theorem 18).
- We establish that, contrary to the ω-regular setting [4], this fragment is not maximal (Proposition 21), and delineate a fragment (Sec. 4.1) that is maximal among properties without greatest fixed points (Theorem 26). We show that the latter fragment is however not maximal in general, and that there is no maximal monitorable fragment whose membership is decidable, in the sense that one could decide whether an input formula can be written in that fragment (Corollary 29). To our minds, those results are the most original contributions of the paper.

Due to space limitations, many proofs are omitted, but can be found in the full version on arXiv.

Related work. Runtime verification tools often integrate some data capabilities. Indeed, according to Falcone *et al.* [34], 13 of the 20 tools surveyed have some data in the input specification. Among tools with data support, we mention AspectJ [8], with data included in regular expression matching, the MOP Framework, which integrates runtime verification with data-handling capabilities into the software development cycle [47]. Rule-based monitor Ruler [13] and the corresponding logic Eagle [12] have both been extended with data parameters. The work [28] uses SMT solvers to handle data added to the (potentially infinite state) monitor directly. Trace slicing reduces the problem to checking projections of traces onto a finite set of values [23] while quantified event automata allow for initial quantification over the domain and then spawn copies of the automaton for all possible values [11]. Finally, the work in [42] studies how to efficiently handle the unbounded memory needs induced by the manipulation of data values in the Dejavu tool, using binary decision diagrams [22] among other techniques.

Another approach is to add data to the logic and monitor fragments thereof. The study in [15] proposes monitors for security policies expressed in metric first order temporal logic. Temporal Object Property Language is a high level logic designed for Java developers, with register automata as a backend formalism [40], bridging the programmer–automata gap.

On the theoretical side, in [16] Bauer et al. study the monitorability of LTL^{FO}, LTL with first order quantification over data. The prefix problem is undecidable, so there is no hope of computing complete monitors but the authors establish a hierarchy based on how much of the trace must be stored. Regarding specifications, the relations between the many logics and automata handling data [27] remain largely unmapped, and most models are not equivalent. Among automata models, register automata are well studied [18]. Pebble automata [49] are closer to logic, but at the cost of decidability. Class memory automata and data automata coincide [17]. Among logics, LTL has been extended to data domains in various ways [27]. In particular, *freeze* LTL is recognisable by alternating register automata [29], which are also closely related to an extension of the modal μ-calculus [46]. We also mention the Logic of Repeating Values [38] and first-order two-variable logic [19] which both have promising algorithmic properties. Beyond the equality predicate, some logics also handle richer domains such as uninterpreted functions [39], and $(\mathbb{Q}, <)$ – the latter is closely related to timed formalisms [37], where the infinity of the alphabet stems from real-valued timestamps [44].

2 The Logic μHML^d

In this section, we define μHML^d, an extension of μHML tailored to express properties of traces of system executions that contain data values. It is a reformulation of the one in [2], which improves it by bringing a more explicit and versatile handling of data through quantifiers and data valuations instead of bindings and substitutions. The former increases expressiveness by allowing one to *guess* data values, while the latter clarifies how data values interact with fixed points. Note that μHML comes in two flavours: *branching time* (the logic describes possible executions of the system) and *linear time* (it describes actual traces). Here, we are concerned with the linear-time setting.

2.1 Data Words and Traces

In formal methods, data words and trees constitute popular formalisms to model respectively the traces and possible executions of systems [51]. Since we consider linear-time properties, we model execution *traces* as data ω-words. They consist in infinite words whose elements

are pairs of a letter from a finite alphabet and of a *data value* from an infinite domain. The finite alphabet plays no role here and can be simulated, so we omit it for simplicity [49]. A data word is thus an infinite sequence of values from an infinite domain.

For the rest of the paper, we fix a countably infinite *data domain* $\mathbb{D}$, whose only predicate is "=" and is decidable. An *action* is modelled as a data value $d \in \mathbb{D}$. An infinite (respectively, finite) *trace* is a data word, *i.e.,* an infinite (resp., finite) sequence $t \in \mathbb{D}^\omega$ (resp., $w \in \mathbb{D}^n$ for some $n \in \mathbb{N}$); the set of all infinite traces is denoted $\text{TRC} = \mathbb{D}^\omega$ (resp., $\text{FTRC} = \mathbb{D}^*$ for finite traces). For $w = w_0 \ldots w_n \in \text{FTRC}$ and $u = u_0 u_1 \cdots \in \text{FTRC} \cup \text{TRC}$, the *concatenation* of w and u is $w \cdot u = w_0 \ldots w_n u_0 \ldots$ (we may omit the $\cdot$). When $u = y \cdot v$, y is a *prefix* of u, and v is a *suffix* of u. The set of suffixes of u is denoted $\texttt{suffix}(u)$.

2.2 Syntax and Semantics

We define an extension of μHML, called μHML^d. Its syntax and semantics are described in Fig. 1 on page 5. Formulae are built from a countable set of formula variables, $X, Y \in \text{FVAR}$, and data variables, $x, y \in \text{DVAR}$, ranging over an infinite domain of data values, $d \in \mathbb{D}$. In addition to the standard Boolean constructs, μHML^d can express recursive properties as least ($\mathsf{min} X.(\varphi)$) and greatest ($\mathsf{max} X.(\varphi)$) fixed-point formulae that bind the free occurrences of X in φ. The logic includes the *possibility* ($\langle b \rangle \varphi$) and *necessity* ($[b]\varphi$) modal constructs. To reason on the data carried by process actions, modalities are augmented with decidable, quantifier-free Boolean *constraint expressions*, $b, c \in \text{BEXP}$, defined over $\mathbb{D}$ and $\text{DVAR} \cup \{\star\}$, where $\star \notin \text{DVAR}$ is a placeholder variable for the current action $d \in \mathbb{D}$. The free data variables $x \in \text{DVAR}$ that appear in b are bound by existential and universal quantification constructs $\exists \boldsymbol{x}.\varphi$ and $\forall \boldsymbol{x}.\varphi$.

In what follows, the standard notions of open and closed expressions, free and bound variables, and formula equivalence up to alpha-conversion are used. We assume wlog that every occurrence of each fixed-point variable is within the scope of a modal operator in its defining fixed-point formula. This is the case e.g. of $\mathsf{max} X.([b]X)$, but not of $\mathsf{max} X.(X \wedge [b]X)$.

We define the domains $\text{DENV} = \text{DVAR} \rightharpoonup \mathbb{D}$ of data environments, $\text{DINT} = \text{DENV} \rightharpoonup 2^{\text{TRC}}$ of data interpretations, and $\text{TENV} = \text{FVAR} \rightharpoonup \text{DINT}$ of trace environments (where $A \rightharpoonup B$ denotes the set of partial functions from set A to set B). A *data environment*, $\delta \in \text{DENV}$, is a partial function with a finite domain mapping data variables to values from $\mathbb{D}$; analogously, a *trace environment*, $\rho \in \text{TENV}$, maps formula variables to data interpretations $F, G \in \text{DINT}$, that given δ, return a set of traces, whose intended meaning is the interpretation of the formula variable in the data environment δ.

The *linear-time* semantics of μHML^d is given by the denotational semantic function, $[\![-]\!]$, defined inductively in Fig. 1. In $[\![-]\!]$, formulae are interpreted w.r.t. a trace environment ρ that gives meaning to formula variables, and a data environment δ that assigns values to data variables in Boolean constraint expressions. Expressions are of the form:

$$b, c \in \text{BEXP} ::= \mathsf{true} \mid e = f \mid \neg b \mid b \wedge c \tag{1}$$

where $e, f \in \text{DVAR} \sqcup \{\star\}$. An expression b defines a set of *external* system actions. An action d is in this set when the data value it carries satisfies b with regards to the data environment δ, *i.e.,* $b\delta[\star \mapsto d] \Downarrow \mathsf{true}$, where, for any function $f : A \to B$, any a (not necessarily in A) and any $b \in B$, $f[a \mapsto b]$ denotes the function that maps a to b and agrees with f over $A \backslash \{a\}$.

μHML^d Syntax. $\varphi,\psi \in \mu\mathrm{HML}^d ::= \mathsf{tt} \mid \mathsf{ff} \mid \langle b\rangle\varphi \mid [b]\varphi \mid \exists \boldsymbol{x}.\varphi \mid \forall \boldsymbol{x}.\varphi \mid \varphi\vee\psi \mid \varphi\wedge\psi \mid \mathsf{min}\, X.(\varphi) \mid \mathsf{max}\, X.(\varphi) \mid X$

Fragments.
$$\varphi,\psi \in \mathrm{cHML}^d ::= \mathsf{tt} \mid \langle b\rangle\varphi \mid \exists \boldsymbol{x}.\varphi \mid \varphi\vee\psi \mid \varphi\wedge\psi \mid \mathsf{min}\, X.(\varphi) \mid X$$
$$\varphi,\psi \in \mathrm{sHML}^d ::= \mathsf{ff} \mid [b]\varphi \mid \forall \boldsymbol{x}.\varphi \mid \varphi\vee\psi \mid \varphi\wedge\psi \mid \mathsf{max}\, X.(\varphi) \mid X$$
$$\varphi,\psi \in \mathrm{disjHML}^d ::= \mathsf{tt} \mid \langle b\rangle\varphi \mid \exists \boldsymbol{x}.\varphi \mid \varphi\vee\psi \mid \mathsf{min}\, X.(\varphi) \mid X$$
$$\varphi,\psi \in \mathrm{HML}^d ::= \mathsf{tt} \mid \mathsf{ff} \mid \langle b\rangle\varphi \mid [b]\varphi \mid \exists \boldsymbol{x}.\varphi \mid \forall \boldsymbol{x}.\varphi \mid \varphi\vee\psi \mid \varphi\wedge\psi$$

Semantics.
$$[\![\mathsf{tt},\rho,\delta]\!] \triangleq \textsc{Trc} \qquad [\![\mathsf{ff},\rho,\delta]\!] \triangleq \varnothing \qquad [\![X,\rho,\delta]\!] \triangleq \big(\rho(X)\big)(\delta)$$
$$[\![\langle b\rangle\varphi,\rho,\delta]\!] \triangleq \big\{t \mid (\exists u,d.t = du \text{ and } b\delta[\star\mapsto d] \Downarrow \mathsf{true} \text{ and } u \in [\![\varphi,\rho,\delta]\!])\big\}$$
$$[\![[b]\varphi,\rho,\delta]\!] \triangleq \big\{t \mid (\forall u,d.(t = du \text{ and } b\delta[\star\mapsto d] \Downarrow \mathsf{true}) \text{ implies } u \in [\![\varphi,\rho,\delta]\!])\big\}$$
$$[\![\exists \boldsymbol{x}.\varphi,\rho,\delta]\!] \triangleq \bigcup_{d\in\mathbb{D}} [\![\varphi,\rho,\delta[x\mapsto d]]\!] \qquad [\![\forall \boldsymbol{x}.\varphi,\rho,\delta]\!] \triangleq \bigcap_{d\in\mathbb{D}} [\![\varphi,\rho,\delta[x\mapsto d]]\!]$$
$$[\![\varphi\vee\psi,\rho,\delta]\!] \triangleq [\![\varphi,\rho,\delta]\!] \cup [\![\psi,\rho,\delta]\!] \qquad [\![\varphi\wedge\psi,\rho,\delta]\!] \triangleq [\![\varphi,\rho,\delta]\!] \cap [\![\psi,\rho,\delta]\!]$$
$$[\![\mathsf{min}\, X.(\varphi),\rho,\delta]\!] \triangleq \Big(\bigsqcap \big\{F \mid \lambda\delta'.[\![\varphi,\rho[X\mapsto F],\delta']\!] \sqsubseteq F\big\}\Big)(\delta)$$
$$[\![\mathsf{max}\, X.(\varphi),\rho,\delta]\!] \triangleq \Big(\bigsqcup \big\{F \mid F \sqsubseteq \lambda\delta'.[\![\varphi,\rho[X\mapsto F],\delta']\!]\big\}\Big)(\delta)$$

Figure 1 Syntax and linear-time semantics of $\mu\mathrm{HML}^d$.

Satisfaction of an expression is then defined as:

$$b\delta \triangleq \begin{cases} \mathsf{true} & b = \mathsf{true} \\ (e\delta) = (f\delta) & b = (e = f) \\ \neg(c\delta) & b = \neg c \\ (c'\delta)\wedge(c''\delta) & b = c'\wedge c'' \end{cases} \qquad e\delta \triangleq \begin{cases} \delta(x) & e = x \text{ and } x \in \mathsf{dom}(\delta) \\ x & e = x \text{ and } x \notin \mathsf{dom}(\delta) \end{cases} \tag{2}$$

Possibility formulae $\langle b\rangle\varphi$ denote all the traces $t = du$ that begin with an action d that is in the action set described by $b\delta$ *and* whose tail u satisfies the continuation formula φ. Dually, necessity formulae $[b]\varphi$ describe all the traces that, *whenever* they begin with such an action d, continue with a trace that satisfies φ. Note that in the linear-time setting, necessity can be expressed as possibility: $[b]\varphi \equiv \langle\neg b\rangle\mathsf{tt}\vee\langle b\rangle\varphi$, and dually $\langle b\rangle\varphi = [\neg b]\mathsf{ff}\wedge[b]\varphi$. The existential quantifier $\exists \boldsymbol{x}.\varphi$ is interpreted as the set of traces that satisfy φ by assigning *some* $d \in \mathbb{D}$ to x; the universal quantifier $\forall \boldsymbol{x}.\varphi$ is the set of traces satisfying φ under *all* such assignments. Formulae are only interpreted with regards to data environments whose domain includes the set of free data variables occurring in them. Note that existential quantification cannot be expressed using universal quantification, except using negation, which is not allowed in the syntax outside modalities.

Since the logic does not have an explicit negation operator, for all φ the semantic function $[\![\varphi,\rho,\delta]\!]$ is monotonic in ρ over the complete lattice $(\mathrm{DInt},\sqsubseteq)$, where the partial order $\sqsubseteq$ corresponds to graph inclusion. Formally, it is defined, for all $F,G \in \mathrm{DInt}$, as $F \sqsubseteq G$ whenever $\forall\delta \in \mathrm{DEnv}.F(\delta) \subseteq G(\delta)$. As is standard in the modal μ-calculus, recursion is interpreted through fixed points: by the Knaster-Tarski theorem [53], $\mathsf{min}\, X.(\varphi)$ and $\mathsf{max}\, X.(\varphi)$ respectively correspond to the least and greatest fixed point of the operator that maps a data interpretation $F : \mathrm{DEnv} \to 2^{\mathrm{Trc}}$ to the data interpretation $\delta \mapsto [\![\varphi,\rho[X\mapsto F],\delta]\!]$. This is the analogue of the operator used to define the semantics of the modal μ-calculus over traces, lifted to the case of infinite alphabets by parameterising the interpretation by a

data environment, in the spirit of [41]. To obtain the sought interpretation for $\mathsf{min}\,X.(\varphi)$ and $\mathsf{max}\,X.(\varphi)$, one then applies the least (resp., greatest) fixed point of this operator (which is a function from data environments to sets of traces) to the current data environment δ.

By construction, they both satisfy the following fixed-point equations where, for all formulae $\varphi,\psi\in\mu\mathrm{HML}^d$ and all recursion variables $X\in\mathrm{FVAR}$, we write $\varphi[\psi/_X]$ for the formula that results by the standard capture-avoiding substitution of all free occurrences of X in φ with ψ:

▶ **Proposition 1.** *For all formulae φ, all trace environments X, all data environments δ, $[\![\mathsf{min}\,X.(\varphi),\rho,\delta]\!]=[\![\varphi[^{\mathsf{min}\,X.(\varphi)}/_X],\rho,\delta]\!]$ and $[\![\mathsf{max}\,X.(\varphi),\rho,\delta]\!]=[\![\varphi[^{\mathsf{max}\,X.(\varphi)}/_X],\rho,\delta]\!]$.*

When a formula is closed with regards to recursion variables (respectively, data variables), its interpretation does not depend on the trace environment ρ (resp., the data environment δ) and we write $[\![\varphi,\delta]\!]$ (resp., $[\![\varphi,\rho]\!]$) in lieu of $[\![\varphi,\rho,\delta]\!]$. For closed formulae, we drop both and write $[\![\varphi]\!]$ in lieu of $[\![\varphi,\rho,\delta]\!]$ for clarity. We say that a trace t satisfies a closed formula φ if $t\in[\![\varphi]\!]$, and violates φ if $t\notin[\![\varphi]\!]$. In the following, in all closed formulae φ we assume that each recursion variable X appears in a unique fixed-point formula $\mathtt{fx}_\varphi(X)$, which is either of the form $\mathsf{min}\,X.(\varphi_X)$ or $\mathsf{max}\,X.(\varphi_X)$. If $\mathtt{fx}(X)$ is $\mathsf{min}\,X.(\varphi_X)$, then X is called an *lfp variable*; otherwise, X is called a *gfp variable*. We write $X\leq Y$ when φ_X is a subformula of φ_Y, $X<Y$ when moreover $X\neq Y$, and denote by $\mathtt{sub}(\varphi)$ the set of subformulae of φ.

▶ **Example 2.** To give an intuition of the logic and its expressiveness, here are a few elementary $\mu\mathrm{HML}^d$ properties, along with their respective fragments:

- The first and second data values are equal (HML^d):

$$\varphi_3 \triangleq \exists\boldsymbol{x}.\langle x=\star\rangle\langle x=\star\rangle\mathsf{tt} \tag{3}$$

  Indeed, the only way for the first modality $\langle x=\star\rangle$ to be satisfied is if x takes the value of the first data value. Then, the second modality $\langle x=\star\rangle$ is satisfied iff the second value is equal to x, hence to the first value.
- The first data value appears again ($\textsc{disj}\mathrm{HML}^d$):

$$\varphi_{\mathrm{leak}} \triangleq \exists\boldsymbol{x}.\langle x=\star\rangle\,\mathsf{min}\,X.(\langle x=\star\rangle\mathsf{tt}\vee\langle x\neq\star\rangle X) \tag{4}$$

  where we use $x\neq\star$ to abbreviate $\neg(x=\star)$. As above, x stores the first data value. Then, we use recursion to look for the second occurrence. Intuitively, on encountering a fixed-point variable X the formula recurses, i.e. we can replace X with the whole $\mathsf{min}\,X.(\varphi)$ that encloses it, as expressed by Proposition 1. Here, the formula recurses while it encounters values satisfying $x\neq\star$, and is satisfied (reaching tt) if it encounters a value satisfying $x=\star$, viz. the first value in the trace. Since this is a least fixed point (min), the formula is violated if it recurses ad infinitum, i.e. if the first value never appears again.
- *Some* data value appears at least twice ($\textsc{disj}\mathrm{HML}^d$):

$$\varphi_5 \triangleq \exists\boldsymbol{x}.\mathsf{min}\,X.(\langle x=\star\rangle\,\mathsf{min}\,Y.(\langle x\neq\star\rangle X\vee\langle x=\star\rangle\mathsf{tt}\vee\langle x\neq\star\rangle Y)) \tag{5}$$

  For a given value of x, the formula accepts only if this value is found once (first disjunct of the first min) and then again (first disjunct of the second, nested min). Overall, the formula accepts whenever there exists such a value, which thus appears twice.
- All data values are pairwise distinct (negation by dualisation of a $\textsc{disj}\mathrm{HML}^d$ formula):

$$\varphi_6 \triangleq \forall\boldsymbol{x}.\mathsf{max}\,X.([x=\star]\,\mathsf{max}\,Y.([x\neq\star]X\wedge[x=\star]\mathsf{ff}\wedge[x\neq\star]Y)) \tag{6}$$

  Dually to the above one, this formula *rejects* whenever some value appears twice.

- The first data value eventually repeats, and in between all data values are pairwise distinct (cHML^d):

$$\varphi_{\text{dist}} \triangleq \exists \boldsymbol{x}.\langle \star = x\rangle \min X.\Big(\langle x = \star\rangle \mathsf{tt} \vee \big(\exists \boldsymbol{y}.\langle \star = y\rangle \min Y.(\langle \star = x\rangle \mathsf{tt} \vee \langle \star \neq x \wedge \star \neq y\rangle Y)\big) \wedge \langle \star \neq x\rangle X\Big)$$

  Here, the first diamond $\langle . \rangle$ implies that x is bound to the first data value. Then, the first $\min$ is satisfied when x occurs again, or when the rhs of the first disjunction is satisfied. This happens when the current data value (bound to y thanks to the $\langle \star = y\rangle$ diamond) does not appear before x is found, as checked by the $\min.Y$, *and* that the overall property is true at next step.
- There exists a data value that never appears (μHML^d):

$$\varphi_7 \triangleq \exists \boldsymbol{x}.\max X.([x = \star]\mathsf{ff} \wedge [x \neq \star]X) \tag{7}$$

  As for φ_6, the $\max$ allows one to forbid a data value (existentially guessed using the $\exists$ quantifier) from appearing in a trace.

2.3 Satisfiability and Validity

Over data words, the infinity of the domain implies that compromises have to be made between expressiveness, closure properties and decidability [17, 27]. By adapting the classical encoding [25, Section 12], one can observe that μHML^d is strictly more expressive than LTL with freeze [30]. Thus, in our setting, decidability fails: the satisfiability and validity problems of μHML^d are undecidable, in contrast with the finite alphabet case (μHML) [54]. By adapting the reduction of [49, Theorem 18], we can sharpen the undecidability result:

▶ **Theorem 3.** *The validity problem for* disjHML^d *is undecidable.*

▶ **Theorem 4.** *The satisfiability problem for* cHML^d *is undecidable.*

The decidability picture for μHML^d is quite grim, but fortunately, as we will see in Sec. 3, this does not prevent us from delineating monitorable fragments of that logic.

2.4 Annotation Semantics

We introduce an alternative semantics for formulae in μHML^d, to better argue about the monitorability of a formula. Annotations are analogous to choice functions [52, Section 4] (see also [24, Theorem 2.1]), and consist in (possibly infinite) witnesses that a formula holds.

▶ **Definition 5.** *An* annotation *is a graph* $(A, \rightarrowtail)$*, where* $A \subseteq \mu HML^d \times DEnv \times Trc$*, and:*

- *it is not the case that* $(\mathsf{ff}, \delta, t) \in A$ *for any* $t \in Trc$ *and* $\delta \in DEnv$*;*
- *if* $(\langle b\rangle\varphi, \delta, dt) \in A$*, then* $b\delta[\star \mapsto d] \Downarrow \mathsf{true}$*,* $(\varphi, \delta, t) \in A$*, and* $(\langle b\rangle\varphi, \delta, dt) \rightarrowtail (\varphi, \delta, t)$*;*
- *if* $([b]\varphi, \delta, dt) \in A$*, and* $b\delta[\star \mapsto d] \Downarrow \mathsf{true}$*, then* $(\varphi, \delta, t) \in A$*, and* $([b]\varphi, \delta, dt) \rightarrowtail (\varphi, \delta, t)$*;*
- *if* $(\exists \boldsymbol{x}.\varphi, \delta, t) \in A$*, then* $(\varphi, \delta[x \mapsto d], t) \in A$ *and* $(\exists \boldsymbol{x}.\varphi, \delta, t) \rightarrowtail (\varphi, \delta[x \mapsto d], t)$ *for some* $d \in \mathbb{D}$*;*
- *if* $(\forall \boldsymbol{x}.\varphi, \delta, t) \in A$*, then* $(\varphi, \delta[x \mapsto d], t) \in A$ *and* $(\forall \boldsymbol{x}.\varphi, \delta, t) \rightarrowtail (\varphi, \delta[x \mapsto d], t)$ *for all* $d \in \mathbb{D}$*;*
- *if* $(\varphi \vee \psi, \delta, t) \in A$*, then* $(\varphi, \delta, t) \in A$ *and* $(\varphi \vee \psi, \delta, t) \rightarrowtail (\varphi, \delta, t)$*, or* $(\psi, \delta, t) \in A$ *and* $(\varphi \vee \psi, \delta, t) \rightarrowtail (\psi, \delta, t)$*;*
- *if* $(\varphi \wedge \psi, \delta, t) \in A$*, then* $(\varphi, \delta, t) \in A$*,* $(\psi, \delta, t) \in A$*,* $(\varphi \wedge \psi, \delta, t) \rightarrowtail (\varphi, \delta, t)$*, and* $(\varphi \wedge \psi, \delta, t) \rightarrowtail (\psi, \delta, t)$*;*
- *if* $(\max X.(\varphi_X), \delta, t) \in A$*, then* $(\varphi_X, \delta, t) \in A$ *and* $(\max X.(\varphi_X), \delta, t) \rightarrowtail (\varphi_X, \delta, t)$*;*

- *if* $(\mathsf{min}\, X.(\varphi_X),\delta,t) \in A$, *then* $(\varphi_X,\delta,t) \in A$ *and* $(\mathsf{min}\, X.(\varphi_X),\delta,t) \rightarrowtail (\varphi_X,\delta,t)$; *and*
- *if* $(X,\delta,t) \in A$, *then* $(\varphi_X,\delta,t) \in A$ *and* $(X,\delta,t) \rightarrowtail (\varphi_X,\delta,t)$.

Given an annotation $(A,\rightarrowtail)$ *and* $X \in FVar$, *we define the relation* $\overset{X}{\rightarrowtail} \subseteq \rightarrowtail$ *thus:* $(\varphi,\delta,t) \overset{X}{\rightarrowtail} (\psi,\delta',u)$ *if and only if* $(\varphi,\delta,t) \rightarrowtail (\psi,\delta',u)$ *and* $\psi \neq Y$ *for any* $Y \in FVar$ *such that* $X < Y$. *We say that* $(A,\rightarrowtail)$ *is* lfp-consistent *if there is no lfp variable* X *that appears infinitely often on a* $\overset{X}{\rightarrowtail}$*-path. For a formula* φ, *a data valuation* $\delta \in DEnv$ *and trace* t, *we say that* $(A,\rightarrowtail)$ *is an annotation for* φ,δ *on* t *(equivalently,* φ,δ *have annotation* $(A,\rightarrowtail)$ *on* t*) if*

1. $A \subseteq \mathtt{sub}(\varphi) \times DEnv \times \mathtt{suffix}(t)$;
2. $(\varphi,\delta,t) \in A$; *and*
3. $(A,\rightarrowtail)$ *is lfp-consistent.*

We say that $(A,\rightarrowtail)$ *is a* finite annotation *for* φ,δ *on* t *when* A *is finite and* $(A,\rightarrowtail)$ *is acyclic.*

Note that all conditions are local, except for lfp-consistency that allows us to distinguish between least and greatest fixed-points by forbidding least fixed-points to be unfolded infinitely often without encountering larger recursion variables.

▶ **Example 6.** Consider the formula φ_{leak} in Equation (4) on trace $u = 0201^\omega$ starting from the empty data valuation. A minimal annotation witnessing that $u \in [\![\varphi_{\text{leak}}]\!]$ is:

$$
\begin{aligned}
&(\exists \boldsymbol{x}.\langle x = \star\rangle \mathsf{min}\, X.((\langle x = \star\rangle \mathsf{tt} \vee \langle x \neq \star\rangle X)),\varnothing,0201^\omega)\\
&\rightarrowtail (\langle x = \star\rangle \mathsf{min}\, X.((\langle x = \star\rangle \mathsf{tt} \vee \langle x \neq \star\rangle X)),x \mapsto 0,0201^\omega)\\
&\rightarrowtail (\mathsf{min}\, X.((\langle x = \star\rangle \mathsf{tt} \vee \langle x \neq \star\rangle X)),x \mapsto 0,201^\omega)\\
&\rightarrowtail (\langle x = \star\rangle \mathsf{tt} \vee \langle x \neq \star\rangle X,x \mapsto 0,201^\omega)\\
&\rightarrowtail (\langle x \neq \star\rangle X,x \mapsto 0,201^\omega) \rightarrowtail (X,x \mapsto 0,01^\omega)\\
&\rightarrowtail (\mathsf{min}\, X.((\langle x = \star\rangle \mathsf{tt} \vee \langle x \neq \star\rangle X)),x \mapsto 0,01^\omega)\\
&\rightarrowtail (\langle x = \star\rangle \mathsf{tt} \vee \langle x \neq \star\rangle X,x \mapsto 0,01^\omega)\\
&\rightarrowtail (\langle x = \star\rangle \mathsf{tt},x \mapsto 0,01^\omega) \rightarrowtail (\mathsf{tt},x \mapsto 0,1^\omega)
\end{aligned}
$$

The following result states that annotations do yield an alternative semantics for μHML^d:

▶ **Proposition 7.** *For every closed* $\varphi \in \mu HML^d$, *all* $\delta \in DEnv$ *and all* $t \in Trc$: φ,δ *have an annotation on* t *if and only if* $t \in [\![\varphi,\delta]\!]$.

Proof sketch. The left-to-right direction follows from the definition, except for fixed points that need to be inductively unfolded using Proposition 1. The lfp-consistency of the annotation allows us to use well-founded induction on the annotation. Conversely, if a trace satisfies a formula, one reconstructs an annotation using the iterative characterisation of fixed points [21, Section 3], in the same spirit as [1, Lemma 2.12 and Theorem 2.13], using transfinite induction to ensure that the constructed annotation is lfp-consistent. ◀

The annotations used in the proof of Proposition 7 may be infinite. However, the only rule that induces cycles or infinite unfolding is max, and only $\forall$ requires infinite branching (and indeed, the below proposition fails for them). Thus, closed formulae in cHML^d that have an annotation on some trace w always admit a finite one. Overall:

▶ **Proposition 8.** *Let* $\varphi \in cHML^d$, $\delta \in DEnv$ *and* $t \in Trc$. *Then,* $t \in [\![\varphi,\delta]\!]$ if and only if φ,δ *have a finite annotation on* t.

▶ **Corollary 9.** *The satisfiability problem for* $cHML^d$ *is recursively enumerable.*

3 Complete and Satisfaction-Complete Fragments

3.1 Monitorability

The goal of this paper is to determine which properties can be verified at runtime. Informally, runtime verification is conducted as follows: along its execution, the *system under scrutiny* produces a trace, whose elements carry information about its operations; it can be thought of as a dynamically produced log file. We do not assume that the system terminates, so the trace is infinite, but termination can obviously be modelled e.g. by a termination symbol.

In parallel with the execution of the system, a program, called *monitor*, passively reads each element of the execution trace in an on-line manner. At any time, the monitor can emit a yes (respectively, a no) verdict, meaning that it considers that the system under scrutiny satisfies (resp., violates) a given specification. We then say that the monitor *accepts* (respectively, *rejects*) the trace. Note, however, that a monitor may never emit a verdict on reading an execution trace, e.g. if it does not have enough information to conclude. In this paper, we focus on *irrevocable* verdicts, meaning that once a verdict is emitted, the monitor cannot change its mind. Thus, for now, we can define a monitor through its acceptance and rejectance predicates (which will later on be defined through an operational model). Our definitions are inspired from [4, 6], although similar ideas already appeared earlier in the literature [50].

▶ **Definition 10** ([4, Definition 3.2 and Theorem 4.8], [6, Definitions 3.1 and 3.4]). *A monitor is an object m on which two predicates $\mathbf{acc}(m,w)$ and $\mathbf{rej}(m,w)$ are defined for all finite traces $w \in F\mathrm{TRC}$, which satisfy the following properties:*
Consistency: *There is no finite trace w such that both $\mathbf{acc}(m,w)$ and $\mathbf{rej}(m,w)$ hold;*
Irrevocability: *For all $w,y \in F\mathrm{TRC}$ such that $w \preceq y$, $\mathbf{acc}(m,w) \Rightarrow \mathbf{acc}(m,y)$ and $\mathbf{rej}(m,w) \Rightarrow \mathbf{rej}(m,y)$.*
We extend the definitions to infinite traces: for all $t \in \mathrm{TRC}$, $\mathbf{acc}(m,t)$ iff there exists some $w \prec t$ such that $\mathbf{acc}(m,w)$, and similarly for $\mathbf{rej}(m,t)$. Finally, a (finite or infinite) trace $u \in F\mathrm{TRC} \cup \mathrm{TRC}$ is accepted *(respectively,* rejected*) by m whenever $\mathbf{acc}(m,u)$ (resp., $\mathbf{rej}(m,u)$).*

Note that the irrevocability criterion implies that monitors only recognise suffix-closed languages, in the sense that both the sets of accepted and rejected traces of a monitor are suffix-closed. We can now relate monitors and properties:

▶ **Definition 11** ([4, Definition 4.1], [6, Definitions 3.3 and 3.5]). *Let $T \subseteq \mathrm{TRC}$ be a property of traces, and m be a monitor.*

We say that m is sound *for satisfactions (respectively, for violations) for T if for all $t \in \mathrm{TRC}$, $\mathbf{acc}(m,t) \Rightarrow t \in T$ (respectively, $\mathbf{rej}(m,t) \Rightarrow t \notin T$). We say that m is* sound *when it is sound for both satisfactions and violations.*

Conversely, we say that m is complete *for satisfactions (resp., violations) for T if the converse holds,* i.e., *for all $t \in \mathrm{TRC}$, $t \in T \Rightarrow \mathbf{acc}(m,t)$ (resp., $t \notin T \Rightarrow \mathbf{rej}(m,t)$). We then say that T is* completely monitorable for satisfactions *(resp.,* for violations*). We say that m is* complete *if it is complete for both, and correspondingly that T is* completely monitorable*.*

We say that the above are effective *when m can be computed by a Turing machine.*

We extend those definitions to any formula $\varphi \in \mu HML^d$ by considering $T = [\![\varphi]\!]$.

In plain words, a property is completely monitorable if there exists a monitor that detects all its satisfactions and violations. Monitorability is thus defined relative to a monitor model, since it depends on the computational power of the monitoring program. As a first step, we consider monitors with arbitrary power; we do not even assume that they are computable.

This very strong definition is to be thought of as an overapproximation. However, as witnessed by Theorem 14, a quite weak monitor model suffices, since completely monitorable properties turn out to be very simple. This motivates the study of satisfaction-completeness in Secs. 3.3 and 4, as well as that of optimal monitors (Definition 13 below) in Sec. 3.4.

For now, a monitor is to be conceived simply as a machine (possibly with access to arbitrary oracles) m which processes a trace and possibly eventually raises a yes or a no verdict. When monitoring for some formula $\varphi \in \mu\text{HML}^d$, if m emits yes upon reading a finite trace $w \in \text{FTRC}$, it means that any continuation $wt \in \mathbb{D}^\omega$ belongs to $[\![\varphi]\!]$. Conversely, if it emits a no, it means that $w\mathbb{D}^\omega \cap [\![\varphi]\!] = \varnothing$. Thus, as long as we are not concerned with the way it executes, a monitor m is fully described by the set of prefixes for which it emits a verdict. Observe that T is completely monitorable iff there exist two sets $G,B \subseteq \text{FTRC}$ such that $T = G\mathbb{D}^\omega = \text{TRC} \setminus (B\mathbb{D}^\omega)$, *i.e.,* it is characterised by its good and bad prefixes.

▶ **Definition 12** ([9, 26]). *Let $T \subseteq \text{TRC}$ be a set of traces. We say that $w \in F\text{TRC}$ is a* good *(respectively, a* bad*) prefix for T when $w\mathbb{D}^\omega \subseteq T$ (respectively, $w\mathbb{D}^\omega \cap T = \varnothing$).*

▶ **Definition 13** ([5, Definition 10]). *Let $T \subseteq \text{TRC}$ be a property of traces and MON be a set of monitors. A monitor $m \in \text{MON}$ is* optimal for violations *(respectively, for satisfactions) in MON for T if for each monitor $m' \in \text{MON}$ that is sound for violations (resp., for satisfactions) for T and each $t \in \text{TRC}$, if $\textbf{\textit{rej}}(m',t)$ then $\textbf{\textit{rej}}(m,t)$ (resp., if $\textbf{\textit{acc}}(m',t)$ then $\textbf{\textit{acc}}(m,t)$).*

If one considers arbitrary monitors (including non-computable ones), we can then say that a monitor m is violation-optimal *for T if for all $w \in F\text{TRC}$, if w is a bad prefix for T then $\textbf{\textit{rej}}(m,w)$, and dually for satisfaction-optimality.*

3.2 The Complete Fragment: HML^d

In the finite alphabet case, all completely monitorable formulae can be expressed in the fragment HML, which consists in formulae of μHML without recursion [4, Theorem 4.8]. The proof of that result can be adapted to establish the following theorem, taming the infinity of the domain by quotienting finite traces by bijections over $\mathbb{D}$ (HML^d is defined in Fig. 1).

▶ **Theorem 14.** *Let $T \subseteq \text{TRC}$ be a set of traces that is stable under renamings (*i.e., *for all bijections $\sigma : \mathbb{D} \to \mathbb{D}$, $\sigma(T) = T$). T is completely monitorable* iff *it can be expressed in HML^d.*

▶ Remark 15. This result does not hold if we consider the domain $(\mathbb{N}, <)$. Indeed, there, one can define the set $D = \{d_0 d_1 \ldots d_n \# w \mid \forall i < j, d_i > d_j\}$, which is completely monitorable, since n is bounded by d_0, but cannot be expressed in HML^d since n depends on d_0.

3.3 A Satisfaction-Complete Fragment: cHML^d

Having to detect *all* satisfactions and *all* violations prevents us from monitoring for behaviours that can happen after an unbounded number of steps in a system execution, restricting us to the tiny fragment HML^d, where properties cannot be recursive. In the following, we relax our notion of completeness and focus on detecting only one kind of verdict, *i.e.,* single-verdict monitors. We also consider the richer setting of optimal monitors in Sec. 3.4, but most results are unfortunately negative. Without loss of generality, we consider satisfaction-completeness, the ability to detect all satisfactions of a property. In practice, as reflected in the literature, runtime verification is more focussed on detecting violations, which are often more critical. Since this adds one level of negation and hence of technicality, we work with satisfactions and results about violation-completeness are obtained by duality.

Syntax. $m,n \in \textsc{Mon} ::= \mathsf{yes} \mid \mathsf{end} \mid (b).m \mid \mathsf{guess}\ x.m \mid m \oplus n \mid m \otimes n \mid \mathsf{rec}\, X.m \mid X$

Configurations. $c \in C ::= (m,\delta) \mid c \odot c$, where $m \in \textsc{Mon}$ is a monitor, $\delta \in \textsc{DEnv}$ is a data environment and $\odot$ is either $\oplus$ (parallel OR) or $\otimes$ (parallel AND).

Small-Step Semantics.

$$\textsc{mVrd}\ \frac{v \in \{\mathsf{yes},\mathsf{end}\}}{v,\delta \xrightarrow{d} v,\delta} \qquad \textsc{mAct}\ \frac{b\delta[\star \mapsto d] \Downarrow \mathsf{true}}{(b).m,\delta \xrightarrow{d} m,\delta}\ d \in \mathbb{D} \qquad \textsc{mBlc}\ \frac{b\delta[\star \mapsto d] \Downarrow \mathsf{false}}{(b).m,\delta \xrightarrow{d} \mathsf{end},\delta}\ d \in \mathbb{D}$$

$$\textsc{mGs}\ \frac{}{\mathsf{guess}\ x.m,\delta \xrightarrow{\tau} m,\delta[x \mapsto d]}\ d \in \mathbb{D} \qquad \textsc{mRec}\ \frac{}{\mathsf{rec}\, X.m,\delta \xrightarrow{\tau} m[\mathsf{rec}\, X.m/_X],\delta}$$

$$\textsc{mFork}\ \frac{}{m \odot n,\delta \xrightarrow{\tau} m,\delta \odot n,\delta} \qquad \textsc{mSyn}\ \frac{c_1 \xrightarrow{d} c_1' \qquad c_2 \xrightarrow{d} c_2'}{c_1 \odot c_2 \xrightarrow{d} c_1' \odot c_2'}$$

$$\textsc{mAsync}_{\mathrm{L}}\ \frac{c_1 \xrightarrow{\tau} c_1'}{c_1 \odot c_2 \xrightarrow{\tau} c_1' \odot c_2} \qquad \textsc{mAsync}_{\mathrm{R}}\ \frac{c_2 \xrightarrow{\tau} c_2'}{c_1 \odot c_2 \xrightarrow{\tau} c_1 \odot c_2'}$$

$$\textsc{mVrC1}\ \frac{}{\mathsf{yes},\delta \otimes c \xrightarrow{\tau} c} \qquad \textsc{mVrD2}\ \frac{}{\mathsf{yes},\delta \oplus c \xrightarrow{\tau} \mathsf{yes},\delta}$$

Synthesis.

$$(\!|\mathsf{tt}|\!) = \mathsf{yes} \qquad (\!|\exists \boldsymbol{x}.\varphi|\!) = \mathsf{guess}\ x.(\!|\varphi|\!) \qquad (\!|\langle b \rangle \varphi|\!) = (b).(\!|\varphi|\!)$$
$$(\!|\varphi \vee \psi|\!) = (\!|\varphi|\!) \oplus (\!|\psi|\!) \qquad (\!|\varphi \wedge \psi|\!) = (\!|\varphi|\!) \otimes (\!|\psi|\!) \qquad (\!|\min X.(\varphi)|\!) = \mathsf{rec}\, X.(\!|\varphi|\!) \qquad (\!|X|\!) = X$$

Figure 2 Syntax, small-step semantics and synthesis of monitors.

Monitor Synthesis. In Figure 2, we introduce a model of monitors, along with a synthesis procedure. We now show that it yields sound and satisfaction-complete monitors for formulae in $\textsc{cHML}^d$ (defined in Fig. 1 on page 5). Note that this fragment includes conjunctions, and it can express $\mathsf{ff} \equiv \langle \bot \rangle \mathsf{tt}$ (where $\bot$ stands e.g. for $x \neq x$) and (linear-time) necessity.

To keep track of the value of each data variable, a monitor $m \in \textsc{Mon}$ is equipped with a data environment $\delta \in \textsc{DEnv}$ forming a pair (m,δ). It begins its execution in the context of an initial data environment δ_0, as a single component (m,δ_0). Unless otherwise stated, $\delta_0 = \varnothing$. Note that for closed monitors, the semantics do not depend on δ. Along its execution, a monitor might fork into parallel components. On forking, each component receives a local copy of the parent monitor's data environment (rule $\textsc{mFork}$) and then evolves independently. The only way to recombine components is when (at least) one has raised a verdict. The verdict is then aggregated with the other components following the usual rules of propositional logic, where yes corresponds to $\top$, $\oplus$ to $\vee$ and $\otimes$ to $\wedge$ (rules $\textsc{mVr}$). To simulate existential quantification, a monitor can non-deterministically guess the value of a data variable and store it in its data environment (rule $\textsc{mGs}$). This overwrites a previous valuation if any.

There are two kinds of transitions. Ones of the form $c \xrightarrow{\tau} c'$ are called τ-transitions, and correspond to internal moves of the monitor, that happen without reading any trace elements. Correspondingly, τ is such that for all (finite or infinite) traces $u \in \textsc{FTrc} \cup \textsc{Trc}$, $\tau u = u$. Those of the form $c \xrightarrow{d} c'$, for $d \in \mathbb{D}$, are transitions that *process* an element from the trace.

For two configurations c,c' and a data value d, we write $c \overset{d}{\Longrightarrow} c'$ whenever $c \xrightarrow{\tau}{}^{*} c'' \xrightarrow{d} c''' \xrightarrow{\tau}{}^{*} c'$ for some configurations c'' and c'''. For a finite trace $w = d_0 d_1 \dots d_l$, we then write $c \overset{w}{\Longrightarrow} c'$ whenever $c \overset{d_0}{\Longrightarrow} c_1 \overset{d_1}{\Longrightarrow} c_2 \dots c_l \overset{d_l}{\Longrightarrow} c'$. By a slight abuse of notation, for all $t \in \textsc{Trc}$, we define $\mathbf{acc}(c,t)$ as $c \overset{w}{\Longrightarrow} \mathsf{yes}, \delta'$ for some $\delta' \in \textsc{Env}$ and some $w \prec t$.

▶ **Example 16.** Consider a server that issues identifier tokens. Assume that the first token it issues is its own and should not be leaked, *i.e.,* that the server *does not* satisfy the formula $\varphi_{\text{leak}} = \exists \boldsymbol{x}. \langle x = \star \rangle \mathsf{min}\, X. (\langle x = \star \rangle \mathsf{tt} \vee \langle x \neq \star \rangle X)$ (already encountered in Ex. 2). The procedure of Fig. 2 yields $m_{\text{leak}} \triangleq (\!|\varphi_{\text{leak}}|\!) = \mathsf{guess}\ x.(\star = x).\mathsf{rec}\, X.((\star = x).\mathsf{yes} \oplus (\star \neq x).X)$.

Consider an erroneous execution "1.0.1…." exhibited by the server. m_{leak} starts in configuration $\mathsf{guess}\ x.(\star = x).\mathsf{rec}\, X.((\star = x).\mathsf{yes} \oplus (\star \neq x).X), \varnothing$. Following rule MGS, m_{leak} *internally* selects a concrete value $d \in \mathbb{D}$ for x. Note that such a value is selected over a possibly infinite domain, reminiscent of [10]. Assume it chooses the value 0 for x. On the next step, the system emits 1 and the monitor checks for the guard $(\star = y)$, which does not hold. Following rule MBLC, it transitions to the inconclusive verdict $\mathsf{end}, x \mapsto 1$, where it stays forever. Assume instead that the monitor picks $x = 1$. Then, we have: $m_{\text{leak}}, \varnothing \xrightarrow[\text{MGS}]{\tau} (\star = x).\mathsf{rec}\, X.((\star = x).\mathsf{yes} \oplus (\star \neq x).X), x \mapsto 1$.

The execution of the monitor continues and it eventually raises a yes verdict. Thus, the trace is accepted by the monitor: it recognises that the system repeats its first action, and hence violates its specification. Note the importance of the non-deterministic choice of a value for x using rule MGS. ⌟

It would not be difficult to establish that m_{leak} is a sound and satisfaction-complete monitor for φ_{leak}. This is more generally the case for CHML^d, and dually for SHML^d:

▶ **Theorem 17.** *CHML^d (respectively, SHML^d) is completely monitorable for satisfactions (resp., violations).*

Proof sketch. Soundness of the synthesis procedure of Fig. 2 is proven similarly to [4, Prop. 4.15]. There, the proof is written for violations but is easily adapted, and data variables do not interfere: they play the same role in monitors as in formulae.

To prove satisfaction-completeness, we use the annotation semantics of Sec. 2.4: in essence, monitors compute annotations of CHML^d formulae, so from an annotation of $\varphi \in \text{CHML}^d$, one can build an accepting run of $(\!|\varphi|\!)$. ◀

Monitors and Register Automata. We conclude by observing that our model of monitors for CHML^d is equivalent to a model of register automata. This result echoes the equivalence between alternation-free modal μ-calculus and register tree automata in [46, Theorems 3 and 7]. With the same ingredients, one can adapt the one in [3, Section 4.2].

Register automata were introduced in [45] as *finite-memory automata*. They consist in a finite-state automaton equipped with a finite set of *registers*, that can store values from an infinite domain (here, $\mathbb{D}$). It is able to compare the value it reads with the content of its registers, and transition accordingly. For a formal definition, see [18, Section 1.3]; one can omit labels which play no role here.

▶ **Theorem 18.** *Let $L \subseteq \mathbb{D}^*$ be a suffix-closed language. There exists an alternating register automaton with existential guessing that recognises L* if and only if *there exists a monitor that accepts exactly the traces in L.*

The correspondence also holds between register automata with no universal (respectively, existential) states and monitors with no $\otimes$ (resp., $\oplus$). Moreover, if one defines $\mathsf{match}(r,b) \triangleq \mathsf{guess}\ r.(b \wedge r = \star)$, all the above correspondences hold for register automata with no guessing and monitors whose $\mathsf{guess}\ r$ construct is replaced with the $\mathsf{match}(r,b)$ one. This implies that a similar equivalence holds between the monitors in [2] and register automata without non-deterministic guessing.

Since all those classes of register automata are inequivalent [18, Section 1.5], we know that all those variants of monitors correspond to different classes of properties. Thus, in $\textsc{cHML}^d$ and $\textsc{sHML}^d$, removing conjunctions or disjunctions reduces expressiveness, and the same holds when replacing existential quantification with a $\mathsf{match}(r,b)$ construct (as defined in [7]). This also shows that deterministic monitors (defined as the counterpart of deterministic register automata) are strictly less expressive than non-deterministic or alternating ones, which invalidates [7, Theorem 18].

3.4 Optimal Monitors

The main obstacle to complete monitorability is that of behaviours that happen in the limit, which obviously cannot be monitored for at runtime. For instance, no monitor can ever detect that there are only finitely many occurrences of a given data value. In the spirit of [5], we thus consider *optimal monitors*, that are only required to flag all violations or satisfactions that may be detected by some monitor.

First, $\textsc{disj}\textsc{HML}^d$ (as defined in Fig. 1) is equivalent to non-deterministic register automata. Their emptiness problem is decidable [45, Theorem 1], so we can build optimal monitors:

▶ **Theorem 19.** *For all $\varphi \in \textsc{disj}\textsc{HML}^d$, one can effectively construct a monitor that is satisfaction-complete and violation-optimal for φ.*

Yet, as soon as we add conjunctions to get $\textsc{cHML}^d$, this becomes impossible. Indeed, from a violation-optimal monitor one can build a semi-algorithm to decide unsatisfiability of $\textsc{cHML}^d$. Since we have a semi-algorithm for satisfiability of $\textsc{cHML}^d$ (Corollary 9), this would yield an algorithm to decide satisfiability of $\textsc{cHML}^d$, contradicting Theorem 4.

▶ **Theorem 20.** *No effective procedure can construct violation-optimal monitors for $\textsc{cHML}^d$.*

4 Satisfaction-Completeness: Beyond cHML^d

We just established that $\textsc{cHML}^d$ is a fragment of $\mu\textsc{HML}^d$ that can be monitored in a sound and satisfaction-complete way with the synthesis procedure of Fig. 2 (Theorem 17), which generalises the finite alphabet case [4, Proposition 4.15]. Moreover, our model of monitors is expressively equivalent to register automata (Theorem 18), which generalises [3, Section 4.2], with the major difference that our monitors cannot be made deterministic.

We now show that, in contrast to the finite alphabet case [4, Proposition 4.18], that fragment is however not maximal: there are properties that admit sound and satisfaction-complete monitors that cannot be expressed in $\textsc{cHML}^d$. Proposition 21 presents such a property, which can be expressed in the larger fragment $\textsc{minHML}^d_{\forall_g}$ that we introduce. The latter is "almost maximal": it is maximal *within* $\textsc{minHML}^d$, *i.e.,* $\mu\textsc{HML}^d$ without greatest fixed-points (Theorem 26), but not in the full $\mu\textsc{HML}^d$. This is for a good reason: there cannot exist a maximally monitorable fragment of $\mu\textsc{HML}^d$ that is effective (Corollary 29).

4.1 A Candidate Maximal Fragment...

In general, universally quantified formulae are not monitorable for satifactions, as they require checking infinitely many instantiations of the quantified variable. Consider, e.g., the formula $\forall \boldsymbol{x}.\mathsf{min}\, X.(\langle \star = x\rangle \mathsf{tt} \vee \langle \star \neq x\rangle X)$ which states that all data values appear in the input. It is satisfiable, since we assumed that the data domain is countable. Yet, it is not monitorable for satisfactions: any finite prefix only contains finitely many data values and can be continued by, e.g., $\#^\omega$, yielding an input which violates the formula.

Nevertheless, some formulae containing universal quantifiers *are* monitorable. Consider the property which states that the input is divided into blocks separated by dollar and sharp symbols, and that all data values that appear in the second block appear in the first block (formalised in Equation 8). It is monitorable for satisfactions: the monitor reads the first two blocks by waiting to see the \$ and then the #; if this never happens it means that the input violates the property. Otherwise, the monitor can check that all data values in the second block appear in the first one by processing them one by one, going back and forth.

$$
\begin{aligned}
&L_{\forall\#\exists\$} = \{d_1 \ldots d_k \$ e_1 \ldots e_l \# \ldots \mid \forall 1 \leq j \leq l, \exists 1 \leq i \leq k, d_i = e_j\}, \text{ expressed as:} \\
&\varphi_{\forall\#\exists\$} = \exists \boldsymbol{x}.\gamma(x) \wedge \forall \boldsymbol{x}.(\gamma(x) \vee \psi(x)), \text{ where:} \\
&\quad \gamma(x) = \mathsf{min}\, X.(\langle \star \neq \$\rangle X \vee \langle \star = \$\rangle \mathsf{min}\, Y.(\langle \star = \#\rangle \mathsf{tt} \vee \langle \star \neq x\rangle Y)) \\
&\quad \psi(x) = \mathsf{min}\, Z.(\langle \star = x\rangle \mathsf{tt} \vee \langle \star \neq \$\rangle Z)
\end{aligned}
\tag{8}
$$

The formula $\gamma(x)$ is called the *guard*, and sets a monitorable bound on the maximal position where a candidate x violating the formula ψ can be found. This way, once the bound is found, the monitor knows that the subsequent data values that appear need not be checked. Here, it expresses that the trace starts with two blocks – ended by \$ and #, respectively – and that x does not appear in the second block: first, look for a \$; once it is found, look for a #, and if in the meantime x is encountered, the formula cannot recurse and therefore rejects.

The formula $\psi(x)$ is the universally quantified property, and since we are looking for satisfactions, its universal quantification has to be limited to finitely many values to ensure that it has a finite witness, hence the disjunction with the guard. Here, it expresses that x appears in the first block. Summing up, the conjunct $\exists \boldsymbol{x}.\gamma(x)$ ensures that a trace has the form $w_1\$w_2\#u$ for some $w_1, w_2 \in \mathbb{D}^*$ and $u \in \textsc{Trc}$, while the conjunct $\forall \boldsymbol{x}.(\gamma(x) \vee \psi(x))$ yields that every $d \in \mathbb{D}$ occurring in w_2 also occurs in w_1.

The above property cannot however be expressed in $\textsc{cHML}^d$. Indeed, the length of w_2 is unbounded, and its elements have to be compared to elements that appear *before* in the input, so they cannot be manipulated only using existential quantifiers, even with fixed points. This is made formal by going through monitors: by Theorem 17, the synthesis procedure of Fig. 2 yields sound a satisfaction-complete monitors. Now, those monitors can only carry boundedly many data values accross the \$ sign. Thus, $\textsc{cHML}^d$ is not a maximally monitorable fragment.

▶ **Proposition 21.** *There does not exist any formula $\varphi \in \textsc{cHML}^d$ such that $[\![\varphi]\!] = L_{\forall\#\exists\$}$.*

We now proceed to characterise the collection of formulae without greatest fixed points that can be monitored in a sound and satisfaction-complete fashion. Given a formula γ, a data variable x, and a finite set $F \subset \textsc{DVar}$ of data variables, we use the following notations:

$$
Fx \triangleq F \cup \{x\}; \quad F\bar{x} \triangleq F \setminus \{x\}; \quad x \neq F \triangleq \bigwedge_{y \in F\bar{x}} \langle x \neq y\rangle \mathsf{tt}; \quad x \sim F \triangleq \bigvee_{y \in F\bar{x}} \langle x = y\rangle \mathsf{tt}; \quad \text{and}
$$

$$
\forall \boldsymbol{x} \leq \boldsymbol{\gamma} + \boldsymbol{F}.\ \varphi \triangleq \exists \boldsymbol{x}.(x \neq F \wedge \gamma) \wedge \forall \boldsymbol{x}.((x \neq F \wedge \gamma) \vee \varphi).
$$

The formula $x \neq F$ describes that the value of x is different from every value assigned to any element of F (except for x if $x \in F$), and $x \sim F$ conversely describes that the value of x coincides with the value assigned to some other element of F.

The quantifier in $\forall \boldsymbol{x} \leq \boldsymbol{\gamma} + \boldsymbol{F}.\ \varphi$ intuitively bounds the quantification of x, as we only need to verify φ for all data values that are assigned to variables in $F\bar{x}$ and the ones for which γ is not true. As such, we say that γ is a guard or bound for x, or that x is guarded. We need to keep track of the free and guarded variables. Hence, we parameterize the definition of our fragment with respect to two finite sets of data variables. For all finite $F \subset \text{DVAR}$ and $V \subseteq F$, we define $\text{MINHML}^d_{\forall_g V,F}$ as the set of formulae that are produced from $\varphi_{V,F}$ in the following grammar whose grammar variables are parameterized with respect to V and F:

$$\varphi_{V,F}, \gamma_{V,F} ::= \mathsf{tt} \mid \mathsf{ff} \mid X \mid \min X.(\varphi_{V,F}) \mid \langle b(F) \wedge \star \neq V \rangle \varphi_{V,F} \mid \varphi_{V,F} \wedge \varphi_{V,F} \mid \varphi_{V,F} \vee \varphi_{V,F}$$
$$\mid \forall \boldsymbol{x} \leq \boldsymbol{\gamma_{Vx,Fx}} + \boldsymbol{F}.\varphi_{V\bar{x},Fx} \mid \exists \boldsymbol{x}.(x \neq V \wedge \varphi_{V\bar{x},Fx}) \vee (x \sim V \wedge \varphi_{Vx,Fx})$$

We then define $\text{MINHML}^d_{\forall_g} = \bigcup_{V \subseteq F \subset \text{DVAR}} \text{MINHML}^d_{\forall_g V,F}$. If $\varphi \in \text{MINHML}^d_{\forall_g}$ has no free data variables, then $\varphi \in \text{MINHML}^d_{\forall_g \varnothing,\varnothing}$. In the above grammar, the set F keeps track of the *free variables* in φ, and V of the "*guarded*" *free variables*. Here, "x is guarded" means that the value of x is not encountered in the trace while we evaluate the formula (but this value is still assigned to some variable). This is ensured by the "$\star \neq V$" conjunct in the diamonds and by guaranteeing that, during the existential quantification of y, if the value of y matches that of some x in V, then y is added to V. Hence $\varphi_{Vx,F}(x)$ can only be true for values of x that do not appear in some finite annotation of φ.

The main characteristic of this fragment is that every universal quantification is *guarded* by a bound on the positions where a candidate x violating the formula can be found. This is achieved by partitioning the potential values of x into those that appear during the (finite) evaluation of the guard (and must be checked against φ) and those that do not (and therefore satisfy the guard). Thus, when monitoring or evaluating $\forall \boldsymbol{x} \leq \boldsymbol{\gamma_{Vx,Fx}} + \boldsymbol{F}.\varphi_{V\bar{x},Fx}$, we only need to consider a fixed number of cases for the value of x when checking the subformula $\gamma_{Vx,Fx}$, and therefore, $\gamma_{Vx,Fx}$ is, in a sense, easier to monitor for, or evaluate, than $\varphi_{V\bar{x},Fx}$. Then, the number of cases that we need to consider for the value of x when checking the subformula $\varphi_{V\bar{x},Fx}$ is finite and depends on how we evaluated $\gamma_{Vx,Fx}$.

In $\varphi_{\forall\#\exists\$}$, the evaluation of the guard γ is complete at the end of the second block. Therefore, to evaluate $\forall x.(\gamma(x) \vee \psi(x))$, it suffices to check values of x for ψ that appear during the evaluation of γ – specifically in the second block. More generally, the grammar of $\text{MINHML}^d_{\forall_g}$ induces a recursive strategy to evaluate a formula while only remembering finitely many cases for the values assigned to its variables. As we see below, this allows us to find a finite witness for the satisfaction of a formula, using a guarded version of annotations, and, subsequently, to monitor for the satisfaction of all formulae in $\text{MINHML}^d_{\forall_g}$.

Guarded-branching Annotations. We can extend the definitions for annotations for CHML^d from Sec. 2.4 to guarded-branching annotations for MINHML^d. For annotation $(A, \rightarrowtail)$, we replace the quantifier conditions with:

if $\boldsymbol{a = (\forall x \leq \gamma + F.\varphi, \delta, t) \in A}$, there is some finite $D \cup \{d_*\} \subseteq \mathbb{D}$ such that $d_* \notin D$, and:

1. $(\gamma, \delta[x \mapsto d_*], t) \in A$ and $a \rightarrowtail (x \neq F \wedge \gamma, \delta[x \mapsto d_*], t)$;
2. for every $d \in D$, $(\varphi, \delta[x \mapsto d], t) \in A$ and $a \rightarrowtail (\varphi, \delta[x \mapsto d], t)$, or $(\gamma, \delta[x \mapsto d], t) \in A$ and $a \rightarrowtail (\gamma, \delta[x \mapsto d], t)$;
3. for every $d \in \mathbb{D}$, having transition $(\gamma, \delta[x \mapsto d_*], t) \rightarrowtail^* (\psi, \delta', du)$ implies $d \in D$; and
4. $\{\delta(x) \mid x \in \text{DVAR}\} \cup (F \cap \mathbb{D}) \subseteq D$.

if $a = (\exists x.(x \neq V \wedge \varphi_1) \vee (x \sim V \wedge \varphi_2), \delta, t)$, then there is some $d \in \mathbb{D}$, such that either
- $d \neq \delta(y)$ for every $y \in V\bar{x}$, and $(\varphi_1, \delta[x \mapsto d], t) \in A$ and $a \rightarrowtail (\varphi_1, \delta[x \mapsto d], t)$; or
- $d = \delta(y)$ for some $y \in V\bar{x}$, and $(\varphi_2, \delta[x \mapsto d], t) \in A$ and $a \rightarrowtail (\varphi_2, \delta[x \mapsto d], t)$.

The condition for the existential quantifier is used to delineate the existential quantifier as it appears in the grammar and the one hidden inside the guarded universal quantifier.

▶ **Theorem 22.** *For every closed $\varphi \in \text{minHML}^d_{\forall_g}$, $\delta \in DEnv$, and $t \in Trc$, $t \in [\![\varphi, \delta]\!]$ if and only if (φ, δ, t) has a finite guarded-branching annotation.*

Proof Sketch. For guarded variables (the ones in V) and for variables whose value does not appear in the annotation, the specific value does not affect the evaluation of the formula, which allows us to show the equivalence of annotations with (finite) guarded-branching ones. For the universal quantifier, D represents the values that we must explicitly consider and $d_\star$ is a "dummy" value that represents all other values. ◀

The Monitorable Least-fixed-point formulae. The guarded-branching annotation semantics for $\text{minHML}^d_{\forall_g}$ yields that the fragment is effectively monitorable for satisfactions, in the sense that satisfactions can be monitored by a Turing machine. The monitors of Sec. 3.3 are equivalent to alternating register automata (Theorem 18), which are computable, so:

▶ **Theorem 23.** *Every formula in cHML^d is effectively monitorable for satisfactions.*

Now, as a consequence of Theorem 22:

▶ **Corollary 24.** *Let $\varphi \in \text{minHML}^d_{\forall_g}$. If $t \in [\![\varphi]\!]$, then t has a good prefix for φ.*

▶ **Corollary 25.** *Every $\varphi \in \text{minHML}^d_{\forall_g}$ is effectively monitorable for satisfactions.*

Moreover, the fragment $\text{minHML}^d_{\forall_g}$ *characterizes* the monitorable properties in minHML^d. There, not all formulae are monitorable, but they are optimally effectively monitorable for satisfactions, in the sense that there exists a satisfaction-optimal monitor for them:

▶ **Theorem 26.** *Every formula $\varphi \in \text{minHML}^d$ is optimally effectively monitorable for satisfactions. A formula $\varphi \in \text{minHML}^d$ is monitorable if and only if $\varphi \equiv \psi$ (*i.e., $[\![\varphi]\!] = [\![\psi]\!]$*) for some $\psi \in \text{minHML}^d_{\forall_g}$.*

To prove this theorem, we introduce $\mathtt{gd}$, that turns every minHML^d formula into a guarded form in $\text{minHML}^d_{\forall_g}$. Let φ be a closed formula without max operators and let $Vr(\varphi) \subseteq \text{DVar}$ be the set of data variables that appear in φ. For every subformula ψ of φ, finite $V \subseteq F \subset Vr(\varphi)$, let $X_{V,F}$ be a new recursion variable associated with X and V, F. For each finite $\Pi \subseteq (2^{Vr(\varphi)})^2$, we define $\mathtt{gd}(\psi, V, F, \Pi)$ by double recursion on $(2^{Vr(\varphi)})^2 \setminus \Pi$ and ψ:
- $\mathtt{gd}(\psi, V, F, \Pi) = \psi$, when $\psi = \mathtt{tt}$ or $\psi = \mathtt{ff}$;
- $\mathtt{gd}(X, V, F, \Pi) = X_{V,F}$, when $(V, F) \in \Pi$;
- $\mathtt{gd}(X, V, F, \Pi) = \mathtt{gd}(\mathtt{fx}(X), V, F, \Pi)$, when $(V, F) \notin \Pi$;
- $\mathtt{gd}(\mathsf{min}X.\psi, V, F, \Pi) = \mathsf{min}X_{V,F}.\mathtt{gd}(\psi, V, F, \Pi \cup \{(V, F)\})$;
- $\mathtt{gd}(\forall x.\psi, V, F, \Pi) = \forall \boldsymbol{x} \leq \mathtt{gd}(\boldsymbol{\psi, Vx, Fx, \Pi}) + \boldsymbol{F}.\mathtt{gd}(\psi, V\bar{x}, Fx, \Pi)$;
- $\mathtt{gd}(\exists x.\psi, V, F, \Pi) = \exists x.(x \neq V \wedge \mathtt{gd}(\psi, V\bar{x}, Fx, \Pi)) \vee (x \sim V \wedge \mathtt{gd}(\psi, Vx, Fx, \Pi))$;
- $\mathtt{gd}(\langle b \rangle \psi, V, F, \Pi) = \langle b \wedge \bigwedge_{x \in V}(x \neq *) \rangle \mathtt{gd}(\psi, V, F, \Pi)$;

and $\mathtt{gd}(-,V,F,\Pi)$ commutes with $\wedge$ and $\vee$. Observe that for all ψ and $V \subseteq Var$, $\mathtt{gd}(\psi,V,F,\Pi) \in \textsc{min}\mathrm{HML}^{d}_{\forall_g V,F}$. We then define $\mathtt{gd}(\psi,V,F) = \mathtt{gd}(\psi,V,F,\varnothing)$ and $\mathtt{gd}(\psi) = \mathtt{gd}(\psi,\varnothing,\varnothing)$, where ψ has no free recursion variables, and, respectively, no free data variables.

The idea behind $\mathtt{gd}$ is to leverage the existence of *good prefixes* for a formula to construct a formula in the guarded fragment. To do so, $\mathtt{gd}$ guards the universal quantification in $\forall x.\psi(x)$ by a "*more monitorable*" formula $\gamma(x)$ that is constructed from $\psi(x)$ by guarding x. Intuitively, a *good prefix* p for $\gamma(x)$ (which exists if the trace is a satisfying one and the formula/guard is monitorable) provides a bound on the part of the trace to consider when looking for candidate values violating $\psi(x)$. Data values outside p are irrelevant: they satisfy $\gamma(x)$ and do not need to be verified against $\psi(x)$.

The operation $\mathtt{gd}$ produces formulae with good monitorability properties when applied to formulae in $\textsc{min}\mathrm{HML}^{d}$. In fact, for each $\varphi \in \textsc{min}\mathrm{HML}^{d}$, $\mathtt{gd}(\varphi) \in \textsc{min}\mathrm{HML}^{d}_{\forall_g}$ and therefore is monitorable for satisfactions. Furthermore, the sound and complete monitor for $\mathtt{gd}(\varphi)$ is *optimal* for φ, in that it can detect all good prefixes for φ; finally, if φ is monitorable for satisfactions, then φ and $\mathtt{gd}(\varphi)$ are equivalent.

4.2 . . . That Is Not Maximal in General

In the finite alphabet case, one can turn greatest fixed points into least fixed points while preserving monitorable consequences by a procedure analogous to determinisation of word automata [5, Section 5]. Over data domains, this is not the case anymore [45, Section 4]. In this section, we show that the addition of max strictly increases the monitorable fragment, and establish that it is undecidable to check if a formula is (effectively) monitorable.

▶ **Lemma 27.** *For each deterministic Turing machine M, we can construct a formula:*
1. *$\psi^{e}_{M} \in \mathrm{sHML}^{d}$, such that $[\![\psi^{e}_{M}]\!]$ is the set of traces that encode the run of M on 0; and*
2. *$\psi^{\neg H}_{M}$, such that $[\![\psi^{\neg H}_{M}]\!]$ is the set of traces that encode a non-empty prefix of a run of M, but do not encode a* terminating *run of M.*

▶ **Corollary 28.** *$\psi^{\neg H}_{T} \in \mu\mathrm{HML}^{d}$ is monitorable for satisfactions, but not effectively monitorable for satisfactions for every T.*

Proof. The formula $\psi^{\neg H}_{T} \in \mu\mathrm{HML}^{d}$ is monitorable for satisfactions, because every satisfying trace t extends a finite trace p that encodes the starting configuration of T on input x. Indeed, if T on x terminates, then every satisfying trace is not a correct encoding of a run of T, and therefore has a good prefix. If T on x does not terminate, then every extension of p satisfies the formula, and therefore p is a good prefix. Therefore, every satisfying t has a good prefix that extends p, yielding that $\psi^{\neg H}_{T} \in \mu\mathrm{HML}^{d}$ is monitorable for satisfactions.

If $\psi^{\neg H}_{T}$ were effectively monitorable, then there would exist a Turing machine M that would recognize the good prefixes of $\psi^{\neg H}_{T}$. M would accept x whenever T does not terminate on x, yielding that the Halting problem is co-recursively enumerable, which is a contradiction. ◀

▶ **Corollary 29.** *Monitorability and effective monitorability for satisfactions for sHML^{d} and $\mu\mathrm{HML}^{d}$ are undecidable.*

Proof. Observe that ψ^{e}_{T} is (effectively) monitorable for satisfactions if and only if T terminates on 0: if T on 0 terminates, then every satisfying trace has a good prefix where an error in the encoding has occurred, or where the full encoding of a run has appeared. Conversely, if T on 0 does not terminate, then the trace that encodes the run of T on 0 has no good prefix, as every prefix can be extended in a way that does not encode the run. ◀

The above result yields the impossibility of a decidable, maximal monitorable fragment of $\mu\mathrm{HML}^{d}$, and similarly for an effectively monitorable fragment.

References

1 Luca Aceto, Antonis Achilleos, Elli Anastasiadi, Adrian Francalanza, and Anna Ingólfsdóttir. Complexity results for modal logic with recursion via translations and tableaux. *Logical Methods in Computer Science*, Volume 20, Issue 3, August 2024. doi:10.46298/lmcs-20(3:14)2024.

2 Luca Aceto, Antonis Achilleos, Duncan Paul Attard, Léo Exibard, Adrian Francalanza, and Anna Ingólfsdóttir. A monitoring tool for linear-time μHML. *Sci. Comput. Program.*, 232:103031, 2024. doi:10.1016/J.SCICO.2023.103031.

3 Luca Aceto, Antonis Achilleos, Adrian Francalanza, Anna Ingólfsdóttir, and Sævar Örn Kjartansson. Determinizing monitors for HML with recursion. *J. Log. Algebraic Methods Program.*, 111:100515, 2020. doi:10.1016/J.JLAMP.2019.100515.

4 Luca Aceto, Antonis Achilleos, Adrian Francalanza, Anna Ingólfsdóttir, and Karoliina Lehtinen. Adventures in monitorability: from branching to linear time and back again. *Proc. ACM Program. Lang.*, 3(POPL):52:1–52:29, 2019. doi:10.1145/3290365.

5 Luca Aceto, Antonis Achilleos, Adrian Francalanza, Anna Ingólfsdóttir, and Karoliina Lehtinen. The best a monitor can do. In Christel Baier and Jean Goubault-Larrecq, editors, *29th EACSL Annual Conference on Computer Science Logic, CSL 2021, January 25-28, 2021, Ljubljana, Slovenia (Virtual Conference)*, volume 183 of *LIPIcs*, pages 7:1–7:23. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.CSL.2021.7.

6 Luca Aceto, Antonis Achilleos, Adrian Francalanza, Anna Ingólfsdóttir, and Karoliina Lehtinen. An operational guide to monitorability with applications to regular properties. *Softw. Syst. Model.*, 20(2):335–361, 2021. doi:10.1007/s10270-020-00860-z.

7 Luca Aceto, Ian Cassar, Adrian Francalanza, and Anna Ingólfsdóttir. On Runtime Enforcement via Suppressions. In *CONCUR*, volume 118 of *LIPIcs*, pages 34:1–34:17, 2018. doi:10.4230/LIPICS.CONCUR.2018.34.

8 Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie J. Hendren, Sascha Kuzins, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to AspectJ. In Ralph E. Johnson and Richard P. Gabriel, editors, *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*, pages 345–364. ACM, 2005. doi:10.1145/1094811.1094839.

9 Bowen Alpern and Fred B. Schneider. Recognizing safety and liveness. *Distributed Comput.*, 2(3):117–126, 1987. doi:10.1007/BF01782772.

10 Krzysztof R. Apt and Gordon D. Plotkin. Countable nondeterminism and random assignment. *J. ACM*, 33(4):724–767, 1986. doi:10.1145/6490.6494.

11 Howard Barringer, Yliès Falcone, Klaus Havelund, Giles Reger, and David E. Rydeheard. Quantified event automata: Towards expressive and efficient runtime monitors. In Dimitra Giannakopoulou and Dominique Méry, editors, *FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings*, volume 7436 of *Lecture Notes in Computer Science*, pages 68–84. Springer, 2012. doi:10.1007/978-3-642-32759-9_9.

12 Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. Rule-based runtime verification. In Bernhard Steffen and Giorgio Levi, editors, *Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004, Venice, Italy, January 11-13, 2004, Proceedings*, volume 2937 of *Lecture Notes in Computer Science*, pages 44–57. Springer, 2004. doi:10.1007/978-3-540-24622-0_5.

13 Howard Barringer, David E. Rydeheard, and Klaus Havelund. Rule systems for run-time monitoring: From Eagle to RuleR. In Oleg Sokolsky and Serdar Tasiran, editors, *Runtime Verification, 7th International Workshop, RV 2007, Vancouver, Canada, March 13, 2007, Revised Selected Papers*, volume 4839 of *Lecture Notes in Computer Science*, pages 111–125. Springer, 2007. doi:10.1007/978-3-540-77395-5_10.

14 Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. Introduction to Runtime Verification. In *Lectures on Runtime Verification: Introductory and Advanced Topics*, volume 10457 of *LNCS*, pages 1–33. Springer, 2018. doi:10.1007/978-3-319-75632-5_1.

15 David A. Basin, Felix Klaedtke, and Samuel Müller. Policy monitoring in first-order temporal logic. In Tayssir Touili, Byron Cook, and Paul B. Jackson, editors, *Computer Aided Verification, 22nd International Conference, CAV2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, volume 6174 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2010. doi:10.1007/978-3-642-14295-6_1.

16 Andreas Bauer, Jan-Christoph Küster, and Gil Vegliach. From propositional to first-order monitoring. In Axel Legay and Saddek Bensalem, editors, *Runtime Verification - 4th International Conference, RV 2013, Rennes, France, September 24-27, 2013. Proceedings*, volume 8174 of *Lecture Notes in Computer Science*, pages 59–75. Springer, 2013. doi:10.1007/978-3-642-40787-1_4.

17 Henrik Björklund and Thomas Schwentick. On notions of regularity for data languages. *Theor. Comput. Sci.*, 411(4-5):702–715, 2010. doi:10.1016/j.tcs.2009.10.009.

18 Mikołaj Bojańczyk. Slightly infinite sets, 2019. URL: https://www.mimuw.edu.pl/~bojan/paper/atom-book.

19 Mikolaj Bojanczyk, Claire David, Anca Muscholl, Thomas Schwentick, and Luc Segoufin. Two-variable logic on data words. *ACM Trans. Comput. Log.*, 12(4):27:1–27:26, 2011. doi:10.1145/1970398.1970403.

20 Benedikt Bollig, Aiswarya Cyriac, Paul Gastin, and K. Narayan Kumar. Model checking languages of data words. In Lars Birkedal, editor, *Foundations of Software Science and Computational Structures*, pages 391–405, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. doi:10.1007/978-3-642-28729-9_26.

21 Julian C. Bradfield and Colin Stirling. Modal mu-calculi. In Patrick Blackburn, J. F. A. K. van Benthem, and Frank Wolter, editors, *Handbook of Modal Logic*, volume 3 of *Studies in logic and practical reasoning*, pages 721–756. North-Holland, 2007. doi:10.1016/S1570-2464(07)80015-2.

22 Randal E. Bryant. Binary decision diagrams. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 191–217. Springer, 2018. doi:10.1007/978-3-319-10575-8_7.

23 Feng Chen and Grigore Rosu. Parametric trace slicing and monitoring. In Stefan Kowalewski and Anna Philippou, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, volume 5505 of *Lecture Notes in Computer Science*, pages 246–261. Springer, 2009. doi:10.1007/978-3-642-00768-2_23.

24 Mads Dam. CTL* and ECTL* as fragments of the modal μ-calculus. *Theoretical Computer Science*, 126(1):77–96, 1994. doi:10.1016/0304-3975(94)90269-0.

25 Mads Dam. Temporal logic, automata and classical theories. In *Proceedings of the 6th European Summer School in Logic, Language and Information (ESSLLI'94)*, August 1994.

26 Marcelo d'Amorim and Grigore Rosu. Efficient monitoring of omega-languages. In Kousha Etessami and Sriram K. Rajamani, editors, *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*, volume 3576 of *Lecture Notes in Computer Science*, pages 364–378. Springer, 2005. doi:10.1007/11513988_36.

27 Loris D'Antoni. In the maze of data languages. *CoRR*, abs/1208.5980, 2012. arXiv:1208.5980.

28 Normann Decker, Martin Leucker, and Daniel Thoma. Monitoring modulo theories. *Int. J. Softw. Tools Technol. Transf.*, 18(2):205–225, 2016. doi:10.1007/S10009-015-0380-3.

29 Stéphane Demri and Ranko Lazic. LTL with the freeze quantifier and register automata. *ACM Trans. Comput. Log.*, 10(3):16:1–16:30, 2009. doi:10.1145/1507244.1507246.

30 Stéphane Demri, Ranko Lazic, and David Nowak. On the freeze quantifier in constraint LTL: decidability and complexity. *Inf. Comput.*, 205(1):2–24, 2007. doi:10.1016/j.ic.2006.08.003.

31 Rüdiger Ehlers, Sanjit A. Seshia, and Hadas Kress-Gazit. Synthesis with identifiers. In Kenneth L. McMillan and Xavier Rival, editors, *Verification, Model Checking, and Abstract*

Interpretation - 15th International Conference, VMCAI 2014, San Diego, CA, USA, January 19-21, 2014, Proceedings, volume 8318 of *Lecture Notes in Computer Science*, pages 415–433. Springer, 2014. doi:10.1007/978-3-642-54013-4_23.

32 E. Allen Emerson. Temporal and modal logic. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 995–1072. Elsevier and MIT Press, 1990. doi:10.1016/b978-0-444-88074-1.50021-4.

33 Léo Exibard, Emmanuel Filiot, and Pierre-Alain Reynier. Synthesis of data word transducers. *Logical Methods in Computer Science*, 17(1), 2021. URL: https://lmcs.episciences.org/7279.

34 Yliès Falcone, Srdan Krstic, Giles Reger, and Dmitriy Traytel. A taxonomy for classifying runtime verification tools. *Int. J. Softw. Tools Technol. Transf.*, 23(2):255–284, 2021. doi:10.1007/S10009-021-00609-Z.

35 Diego Figueira. *Reasoning on words and trees with data. (Raisonnement sur mots et arbres avec données)*. PhD thesis, École normale supérieure de Cachan, France, 2010. URL: https://tel.archives-ouvertes.fr/tel-00718605.

36 Diego Figueira. Alternating register automata on finite words and trees. *Log. Methods Comput. Sci.*, 8(1), 2012. doi:10.2168/LMCS-8(1:22)2012.

37 Diego Figueira, Piotr Hofman, and Slawomir Lasota. Relating timed and register automata. *Math. Struct. Comput. Sci.*, 26(6):993–1021, 2016. doi:10.1017/S0960129514000322.

38 Diego Figueira, Anirban Majumdar, and M. Praveen. Playing with repetitions in data words using energy games. *Log. Methods Comput. Sci.*, 16(3), 2020. URL: https://lmcs.episciences.org/6614.

39 Bernd Finkbeiner, Felix Klein, Ruzica Piskac, and Mark Santolucito. Temporal stream logic: Synthesis beyond the bools. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*, volume 11561 of *Lecture Notes in Computer Science*, pages 609–629. Springer, 2019. doi:10.1007/978-3-030-25540-4_35.

40 Radu Grigore, Dino Distefano, Rasmus Lerchedahl Petersen, and Nikos Tzevelekos. Runtime verification based on register automata. In Nir Piterman and Scott A. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7795 of *Lecture Notes in Computer Science*, pages 260–276. Springer, 2013. doi:10.1007/978-3-642-36742-7_19.

41 Jan Friso Groote and Radu Mateescu. Verification of temporal properties of processes in a setting with data. In Armando Martin Haeberer, editor, *Algebraic Methodology and Software Technology, 7th International Conference, AMAST '98, Amazonia, Brasil, January 4-8, 1999, Proceedings*, volume 1548 of *Lecture Notes in Computer Science*, pages 74–90. Springer, 1998. doi:10.1007/3-540-49253-4_8.

42 Klaus Havelund and Doron Peled. Efficient runtime verification of first-order temporal properties. In María-del-Mar Gallardo and Pedro Merino, editors, *Model Checking Software - 25th International Symposium, SPIN 2018, Malaga, Spain, June 20-22, 2018, Proceedings*, volume 10869 of *Lecture Notes in Computer Science*, pages 26–47. Springer, 2018. doi:10.1007/978-3-319-94111-0_2.

43 Klaus Havelund, Giles Reger, Daniel Thoma, and Eugen Zalinescu. Monitoring events that carry data. In Ezio Bartocci and Yliès Falcone, editors, *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *Lecture Notes in Computer Science*, pages 61–102. Springer, 2018. doi:10.1007/978-3-319-75632-5_3.

44 Hsi-Ming Ho, Joël Ouaknine, and James Worrell. Online monitoring of metric temporal logic. In Borzoo Bonakdarpour and Scott A. Smolka, editors, *Runtime Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings*,

volume 8734 of *Lecture Notes in Computer Science*, pages 178–192. Springer, 2014. doi:10.1007/978-3-319-11164-3_15.

45 Michael Kaminski and Nissim Francez. Finite-memory automata. *Theor. Comput. Sci.*, 134(2):329–363, 1994. doi:10.1016/0304-3975(94)90242-9.

46 Marcin Jurdzinski and Ranko Lazic. Alternation-free modal mu-calculus for data trees. In *22nd IEEE Symposium on Logic in Computer Science (LICS 2007), 10-12 July 2007, Wroclaw, Poland, Proceedings*, pages 131–140. IEEE Computer Society, 2007. doi:10.1109/LICS.2007.11.

47 Patrick O'Neil Meredith, Dongyun Jin, Dennis Griffith, Feng Chen, and Grigore Rosu. An overview of the MOP runtime verification framework. *Int. J. Softw. Tools Technol. Transf.*, 14(3):249–289, 2012. doi:10.1007/S10009-011-0198-6.

48 Robin Milner. Is computing an experimental science? *J. Inf. Technol.*, 2(2):58–66, 1987. doi:10.1057/JIT.1987.12.

49 Frank Neven, Thomas Schwentick, and Victor Vianu. Finite state machines for strings over infinite alphabets. *ACM Trans. Comput. Log.*, 5(3):403–435, 2004. doi:10.1145/1013560.1013562.

50 Amir Pnueli and Aleksandr Zaks. PSL model checking and run-time verification via testers. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006: Formal Methods, 14th International Symposium on Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 573–586. Springer, 2006. doi:10.1007/11813040_38.

51 Luc Segoufin. Automata and logics for words and trees over an infinite alphabet. In Zoltán Ésik, editor, *Computer Science Logic, 20th International Workshop, CSL 2006, 15th Annual Conference of the EACSL, Szeged, Hungary, September 25-29, 2006, Proceedings*, volume 4207 of *Lecture Notes in Computer Science*, pages 41–57. Springer, 2006. doi:10.1007/11874683_3.

52 Robert S. Streett and E. Allen Emerson. An automata theoretic decision procedure for the propositional mu-calculus. *Information and Computation*, 81(3):249–264, 1989. doi:10.1016/0890-5401(89)90031-X.

53 Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955. doi:10.2140/pjm.1955.5.285.

54 Moshe Y. Vardi. A temporal fixpoint calculus. In Jeanne Ferrante and Peter Mager, editors, *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988*, pages 250–259. ACM Press, 1988. doi:10.1145/73560.73582.

Characterizations of Fragments of Temporal Logic over Mazurkiewicz Traces

Bharat Adsul
IIT Bombay, Mumbai, India

Paul Gastin
Université Paris-Saclay, ENS Paris-Saclay, CNRS, LMF, 91190, Gif-sur-Yvette, France
CNRS, ReLaX, IRL 2000, Siruseri, India

Shantanu Kulkarni
IIT Bombay, Mumbai, India

Abstract

We study fragments of temporal logics over Mazurkiewicz traces which are well known and established partial-order models of concurrent behaviours. We focus on concurrent versions of "strict past" and "strict future" modalities. Over words, the corresponding fragments have been shown to coincide with natural algebraic conditions on the recognizing monoids. We provide non-trivial generalizations of these classical results to traces. We exploit the local nature of the temporal modalities and obtain modular translations of specifications into asynchronous automata. More specifically, we provide novel characterizations of these fragments via local cascade products of a very simple two-state asynchronous automaton operating on a single process.

2012 ACM Subject Classification Theory of computation → Modal and temporal logics; Theory of computation → Distributed computing models; Theory of computation → Algebraic language theory; Security and privacy → Logic and verification

Keywords and phrases Mazurkiewicz traces, temporal logics, asynchronous automata, cascade product, Green's relations, algebraic automata theory

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2025.5

1 Introduction

Temporal logics are widely used logical formalisms to reason about properties of computations. Words serve as temporal models of sequential computations/behaviours and linear temporal logic LTL formulas use natural future as well as past temporal modalities to talk about interesting patterns of these behaviours. The celebrated theorem of Kamp [16] showed that, over words, properties definable in LTL coincide with those definable in the first order (FO) logic. By the seminal results of McNaughton-Papert [18] and Schutzenberger [20], these also equal regular languages describable by star-free regular expressions and those recognized by finite aperiodic monoids.

The above mentioned results clearly indicate the important role played by the temporal modalities in the very rich expressive power of LTL. Natural and commonly used temporal modalities include future modalities X ("neXt/tomorrow"), F ("Future"), U ("Until"), the past counterparts Y ("previous/Yesterday"), P ("Past"), S ("Since") and their strict variants such as XF ("strict future") and YP ("strict past"). Several works [7], [14], [5] have studied natural fragments of LTL obtained by allowing only some of these modalities and have characterized these fragments in terms of algebraic conditions on the recognizing finite monoids. These algebraic characterizations have been crucially used to show that the computational problems of definability in the associated fragments are decidable.

Let us consider the fragment $\mathsf{LTL}[\mathsf{YP}]$ obtained by allowing only unary strict past modality YP. When interpreted over words, the formula $\mathsf{YP}\,\varphi$, at a given position, asserts that φ holds at some position in the strict past (that is, strictly left) of the current position. It has

© Bharat Adsul, Paul Gastin, and Shantanu Kulkarni;
licensed under Creative Commons License CC-BY 4.0

been shown in [6], [7], [14], [5] that a language is definable in $\mathsf{LTL}[\mathsf{YP}]$ iff it can be recognized by a $\mathcal{R}$-trivial monoid iff the syntactic monoid of the language is $\mathcal{R}$-trivial. Here $\mathcal{R}$ stands for one of fundamental Green's relations on a monoid. The $\mathcal{R}$-triviality condition on a monoid M simply means that if two elements of M are right multiples of each other then they are equal (that is, for all $m, n \in M$, $\exists\, p, q \in M : n = m \cdot p$ and $m = n \cdot q$ implies $m = n$). This characterization can also be stated in simpler automata-theoretic terms [6]. A finite-state automaton is said to be partially ordered if there is a partial order $\succeq$ on the states so that every transition is of the form (p, a, q) where $p \succeq q$. Said differently, an automaton is partially ordered iff the only cycles present in the underlying directed transition graph are self loops. Using this notion, the above mentioned characterization can be stated as follows: a language is $\mathsf{LTL}[\mathsf{YP}]$-definable iff it can be accepted by a partially ordered automaton.

It turns out that the dual fragment $\mathsf{LTL}[\mathsf{XF}]$ which admits the unary strict future modality XF is characterized by the dual condition of $\mathcal{L}$-triviality. As expected, a monoid is $\mathcal{L}$-trivial if no two distinct elements can be left multiples of each other. The fragment $\mathsf{LTL}[\mathsf{XF}, \mathsf{YP}]$ which includes both XF and YP modalities has been well studied and is known to coincide with important first order logic fragments, namely, FO^2 and Δ_2. It also has an algebraic characterization in terms of monoids from the class DA. See the survey [9] for more details.

In this work we study the above mentioned temporal logic fragments from the concurrency viewpoint. We consider Mazurkiewicz traces as the underlying models of concurrent behaviours. A Mazurkiewicz trace or simply trace, over a set of processes, represents a concurrent behaviour as a labelled partial-order between events. This partial-order between events faithfully captures the *causal* information flow among processes. Traces are extensively studied and there is a rich theory of *regular* trace languages [23], [12]. Regular trace languages are known to coincide with MSO-definable languages as well as languages recognized by finite monoids. We consider natural *event-based* interpretations of temporal modalities XF and YP over traces. For instance, the formula $\mathsf{YP}\ \varphi$, at a given event, asserts that φ holds at an event which is in the strict causal past of the current event. We denote by $\mathsf{LocTL}[\mathsf{YP}]$ the local/event-based temporal logic over traces which has access to the temporal modality YP. One can similarly define $\mathsf{LocTL}[\mathsf{XF}]$ and $\mathsf{LocTL}[\mathsf{XF}, \mathsf{YP}]$. One of our main results states that a trace language is definable in $\mathsf{LocTL}[\mathsf{YP}]$ iff it can be recognized by a $\mathcal{R}$-trivial monoid. This can be viewed as a non-trivial generalization of the corresponding result over words. We also obtain an analogous algebraic characterization of $\mathsf{LocTL}[\mathsf{XF}]$ in terms of $\mathcal{L}$-trivial monoids. An algebraic characterization of $\mathsf{LocTL}[\mathsf{XF}, \mathsf{YP}]$ in terms of monoids from the class DA already appears in [17].

Another important contribution of this work is a *modular* translation of formulas in these fragments into *asynchronous automata*. In an asynchronous automaton, each process runs a finite local-state device and these devices synchronize with each other on shared actions. Asynchronous automata can be naturally used to accept trace languages. Zielonka's theorem states that regular trace languages are precisely the languages accepted by asynchronous automata. As in [19, 2, 3], one can use asynchronous automata as transducers on traces, similar in spirit to the sequential letter-to-letter transducers on words. Unlike these works which use *deterministic* asynchronous transducers to locally compute relabelling functions on input traces, we allow non-deterministic asynchronous transducers. As expected, these transducers compute relabelling relations from input traces to output traces. The *local cascade product* of deterministic asynchronous automata (or transducers) from [2, 3] is a natural generalization of cascade product in the sequential setting [21]. It further extends naturally to the non-deterministic setting and corresponds to compositions of related relabelling relations.

We use this machinery to provide a translation (in fact, a characterization) of $\mathsf{LocTL}[\mathsf{XF}, \mathsf{YP}]$ formulas into local cascade products of very simple two-state asynchronous automata operating on a single process. For $\mathsf{LocTL}[\mathsf{YP}]$ formulas, the resulting atomic two-state transducers are deterministic while for $\mathsf{LocTL}[\mathsf{XF}]$ formulas, they are reverse-deterministic. We remark that our results and their proofs for traces specialize to the setting of words.

The rest of the paper is organized as follows. After setting up the preliminary notions in Section 2, we establish in Section 3 that $\mathcal{R}$-trivial trace languages are definable in $\mathsf{LocTL}[\mathsf{YP}]$. In Section 4, we provide a translation of $\mathsf{LocTL}[\mathsf{YP}]$ formulas into cascade products of *localized* deterministic asynchronous transducers/automata whose transition monoid is isomorphic to U_1 - the unique two element aperiodic monoid. Section 5 is devoted to dual results concerning $\mathcal{L}$-trivial and $\mathsf{LocTL}[\mathsf{XF}]$-definable trace languages. We also sketch a cascade product based characterization of $\mathsf{LocTL}[\mathsf{XF}, \mathsf{YP}]$ in this section.

2 Preliminaries

In this section, we set up the notation and review basics of traces. We also recall the recognizability of trace languages by morphisms into monoids and define the fundamental Green's relations on monoids.

2.1 Trace basics

Let $\mathcal{P}$ be a finite, non-empty set of processes. For every process $i \in \mathcal{P}$, Σ_i is the set of letters/actions in which process i participates. A distributed alphabet over $\mathcal{P}$ is the family $\tilde{\Sigma} = \{\Sigma_i\}_{i\in\mathcal{P}}$ with $\Sigma = \cup_{i\in\mathcal{P}}\Sigma_i$ being the associated total alphabet. Letters $a, b \in \Sigma$ are dependent if there is a process which participates in both of them ($\exists i \in \mathcal{P}$, $a, b \in \Sigma_i$), otherwise they are independent. For a letter $a \in \Sigma$ we define $\mathsf{loc}(a) = \{i \in \mathcal{P} \mid a \in \Sigma_i\}$ to be the set of processes that participate in a.

A poset is a tuple $(E, \leq)$ with E being the underlying set and $\leq$ being the partial order on E. A Σ-labelled poset over $\tilde{\Sigma}$ is a triple $(E, \leq, \lambda)$ where $(E, \leq)$ is the underlying poset, and $\lambda\colon E \to \Sigma$ is the labelling function which labels each element in E with a letter from Σ. For elements $e, e' \in E$, e' is said to be an *immediate successor* of e, denoted by $e \lessdot e'$, if e is below e' ($e < e'$) and there is no event f strictly between e and e' ($\nexists f \in E$, $e < f < e'$).

A (*Mazurkiewicz*) *trace* $t = (E, \leq, \lambda)$ over $\tilde{\Sigma}$ is a finite Σ-labelled poset with λ satisfying the following two properties:

1. for $e, e' \in E$, if $e \lessdot e'$, then $\lambda(e)$ and $\lambda(e')$ are dependent;
2. for $e, e' \in E$, if $\lambda(e)$ and $\lambda(e')$ are dependent, then they are ordered ($e \leq e'$ or $e' \leq e$).

For a trace $t = (E, \leq, \lambda)$ we sometimes slightly abuse notation to say event $e \in t$ which we take to mean that event $e \in E$. The set of process i-events in t is defined as $E_i = \{e \in E \mid \lambda(e) \in \Sigma_i\}$. Note that E_i is totally ordered by $\leq$. For an event $e \in t$, we slightly abuse the notation and define $\mathsf{loc}(e) = \mathsf{loc}(\lambda(e))$ to be the set of processes that participate in e. For an event e we define $\downarrow e = \{f \in t \mid f \leq e\}$ to be the set of events in t below e. Similarly $\Downarrow e = \{f \in t \mid f < e\}$ is the set of events in t strictly below e. It is easy to see that by restricting $\leq$ and λ to events in $\downarrow e$, we obtain a valid trace. By abuse of notation, we denote this trace by $(\downarrow e, \leq, \lambda)$ or simply $\downarrow e$. It will be clear from the context whether we mean $\downarrow e$ to be a trace or merely a set of events. Similarly, $\Downarrow e$ also refers to the trace $(\Downarrow e, \leq, \lambda)$ induced by $\Downarrow e$.

The set of all finite traces over $\tilde{\Sigma}$ is denoted by $TR(\tilde{\Sigma})$. We define the *trace concatenation* operation $\cdot$ on $TR(\tilde{\Sigma})$ as follows. Let $t_1 = (E_1, \leq_1, \lambda_1), t_2 = (E_2, \leq_2, \lambda_2) \in TR(\tilde{\Sigma})$ with $E_1 \cap E_2 = \emptyset$. We define $t_1 \cdot t_2$ or simply $t_1 t_2$ to be the trace $(E_1 \cup E_2, \leq, \lambda) \in TR(\tilde{\Sigma})$ where:
1. $\leq$ is the transitive closure of $\leq_1 \cup \leq_2 \cup \{(e_1, e_2) \in E_1 \times E_2 \mid \lambda_1(e_1), \lambda_2(e_2)$ are dependant$\}$.
2. For all $e \in E_1 \cup E_2$, $\lambda(e) = \lambda_1(e)$ if $e \in E_1$ and $\lambda_2(e)$ if $e \in E_2$.

A trace t' is said to be a prefix of trace t if there exists $t'' \in TR(\tilde{\Sigma})$ such that $t = t't''$. It is easy to see that, for $e \in t$, $\downarrow e$ and $\Downarrow e$ are both prefixes of t.

2.2 Recognizable trace languages

Recall that a monoid is a set M equipped with a binary associative operation $\cdot$ and containing an identity element 1_M, such that $\forall m \in M$, $m \cdot 1_M = 1_M \cdot m = m$. The set $TR(\tilde{\Sigma})$, under the operation of *trace concatenation*, is a monoid with the empty trace ε as the identity element. Let M and N be two monoids with operations $\cdot$ and $\circ$ respectively. A *morphism* h from M to N is a map $h\colon M \to N$ such that: $h(1_M) = 1_N$ and for all $m_1, m_2 \in M$, $h(m_1 \cdot m_2) = h(m_1) \circ h(m_2)$.

A subset $L \subseteq TR(\tilde{\Sigma})$ is called a *trace language* over $\tilde{\Sigma}$. We call a morphism from the trace monoid $TR(\tilde{\Sigma})$ to any other monoid M as a trace morphism. A trace language $L \subseteq TR(\tilde{\Sigma})$ is recognizable if there exists a trace morphism h from $TR(\tilde{\Sigma})$ to a *finite* monoid M such that $L = \cup_{m \in P} h^{-1}(m)$ for some $P \subseteq M$. We may refer to h as the recognizing morphism or say that L is recognized by M via recognizing morphism h. The class of recognizable trace languages, also referred to as *regular trace languages*, has several characterizations and we note a few in particular, namely MSO-definable languages over traces [22] and trace languages accepted by asynchronous automata [23].

Classically, the algebraic viewpoint in terms of what are known as *Green's relations* over monoids has been fruitful for characterizing interesting subclasses of regular languages which have an intimate connection with logics. We now briefly recall these fundamental Green's preorders and the associated equivalence relations.

Let M be a monoid with $\cdot$ as the binary monoid operation and let $m_1, m_2 \in M$. The Green's preorder relations are defined as follows:
1. $m_1 \leq_R m_2$ (read as m_1 is R-below m_2) if there exists $m \in M$ such that $m_1 = m_2 \cdot m$.
2. $m_1 \leq_L m_2$ if there exists $m \in M$ such that $m_1 = m \cdot m_2$.
3. $m_1 \leq_J m_2$ if there exist $u, v \in M$ such that $m_1 = u \cdot m_2 \cdot v$
4. $m_1 \leq_H m_2$ if $m_1 \leq_R m_2$ and $m_1 \leq_L m_2$

These preorders naturally give rise to the associated equivalence relations $\mathcal{R}, \mathcal{L}, \mathcal{J}, \mathcal{H}$. We say that $m_1 \mathcal{R} m_2$ (read as m_1 is $\mathcal{R}$-equivalent to m_2) if $m_1 \leq_R m_2$ and $m_2 \leq_R m_1$. A monoid M is $\mathcal{R}$-trivial if $\forall m_1, m_2 \in M$, $m_1 \mathcal{R} m_2$ implies $m_1 = m_2$. In other words, M is $\mathcal{R}$-trivial iff the preorder $\leq_R$ on M is in fact a partial order on M iff the equivalence relation $\mathcal{R}$ coincides with the identity relation on M.

The other equivalence relations $\mathcal{L}, \mathcal{J}, \mathcal{H}$ and their triviality for M are defined analogously. A trace language $L \subseteq TR(\tilde{\Sigma})$ is called $\mathcal{R}$-trivial if it can be recognized by a finite $\mathcal{R}$-trivial monoid. One can similarly define $\mathcal{L}$-trivial, $\mathcal{J}$-trivial and $\mathcal{H}$-trivial trace languages.

▶ **Example 1.** Consider the monoid $U_1 = \{1, 0\}$ with 1 as the identity and 0 as the zero element. So, $1 \cdot 1 = 1$, $0 \cdot 1 = 1 \cdot 0 = 0 \cdot 0 = 0$. It is easily checked that U_1 is $\mathcal{R}$-trivial, $\mathcal{L}$-trivial, $\mathcal{J}$-trivial as well as $\mathcal{H}$-trivial.

As mentioned in the introduction, for word languages, $\mathcal{R}$-triviality ($\mathcal{L}$-triviality) corresponds to definability in LTL with only the strict past (respectively strict future) modality. See [5] for a proof based on Green's relations. It turns out that, for finite monoids,

$\mathcal{H}$-triviality condition is equivalent to the aperiodicity property. As a consequence, $\mathcal{H}$-trivial word languages coincide with languages definable in the first order logic and linear temporal logic. These results find interesting generalizations over traces in [15, 13, 8].

3 Algebra to temporal logic

We now state our main theorem and set up the technical machinery and prove it.

▶ **Theorem 2.** *Let $L \subseteq TR(\tilde{\Sigma})$. Then the following are equivalent.*

1. *L is a $\mathcal{R}$-trivial trace language.*
2. *L can be expressed using a sentence in* LocTL[YP].
3. *L can be accepted by a cascade product of localized U_1 asynchronous automata.*

Proof. $1 \to 2$ is shown in Theorem 14, $2 \to 3$ is shown in Theorem 22, $3 \to 1$ is shown in Theorem 24. ◀

The result we wish to prove in this section is the $1 \to 2$ for Theorem 2, which says that any trace language which is recognized by a $\mathcal{R}$-trivial monoid can be expressed as a sentence in LocTL[YP]. This statement is restated as Theorem 14. We start by defining formally the syntax and semantics of LocTL[YP].

3.1 LocTL[YP] syntax and semantics

We now define LocTL[YP]. We have sentences Φ, which are evaluated over finite Mazurkiewicz traces and hence define languages over $TR(\tilde{\Sigma})$ and also event formulas φ which are evaluated at a given event in a trace. The syntax is as follows:

$$\Phi ::= \mathsf{E}\,\varphi \mid \Phi_1 \vee \Phi_2 \mid \neg\Phi$$
$$\varphi ::= a \mid \varphi_1 \vee \varphi_2 \mid \neg\varphi \mid \mathsf{YP}\,\varphi$$

The semantics for the boolean operations are as usual. For a trace $t = (E, \leq, \lambda)$ and an event $e \in E$, the following defines the semantics of LocTL[YP]

$t \models \mathsf{E}\,\varphi$	if there exists an event e in t such that $t, e \models \varphi$
$t, e \models a$	if $\lambda(e) = a$
$t, e \models \mathsf{YP}\,\varphi$	if there exists an event f in t with $f < e$ and $t, f \models \varphi$.

In the rest of the paper we shall use the following macros liberally:

$$\mathsf{A}\,\varphi = \neg\,\mathsf{E}\,\neg\varphi, \qquad \mathsf{P}\,\varphi = \varphi \vee \mathsf{YP}\,\varphi,$$
$$\mathsf{YH}\,\varphi = \neg\,\mathsf{YP}\,\neg\varphi, \qquad \mathsf{H}\,\varphi = \varphi \wedge \mathsf{YH}\,\varphi\,.$$

The modalities can be read as Exists (E), All (A), Past (P), Historically (H), Yesterday Past (YP), Yesterday Historically (YH). For a set of letters A, we may abuse notation and use A to denote the event formula $\bigvee_{a\in A} a$ testing whether the event is labelled by a letter in A. For instance, we may use Σ_i as the macro for the event formula $\bigvee_{a\in\Sigma_i} a$ which tests whether the current event is an i-event.

▶ **Example 3.** Let $t \in TR(\tilde{\Sigma})$ be a trace and $e \in t$ be an event. We design an event formula φ^i_k such that $t, e \models \varphi^i_k$ iff there are at least k process i-events strictly below event e. We define φ^i_k inductively as: $\varphi^i_0 = \top$ and for $k > 0$, $\varphi^i_k = \mathsf{YP}(\Sigma_i \wedge \varphi^i_{k-1})$. Consider the LocTL[YP] sentence $\Phi_{m,n} = \mathsf{E}(\Sigma_j \wedge \varphi^j_{n-1} \wedge \neg\varphi^j_n \wedge \varphi^i_m)$. Clearly $\Phi_{m,n}$ has the property that trace t satisfies $\Phi_{m,n}$ if and only if the m^{th} process i-event is strictly below the n^{th} process j-event.

3.2 Critical structures for a morphism into a $\mathcal{R}$-trivial monoid

Throughout the rest of this section, we fix a morphism $h\colon TR(\tilde{\Sigma}) \to M$ to a fixed finite $\mathcal{R}$-trivial monoid M.

Let $t = (E, \leq, \lambda)$ be a trace and $e \in E$ be an event of t. Note that, viewing $\downarrow e$ and $\Downarrow e$ as traces, we have $h(\downarrow e) = h(\Downarrow e)h(\lambda(e)) \leq_R h(\Downarrow e)$. The event e is said to be a *critical event* in t if $h(\downarrow e) \neq h(\Downarrow e)$. Since the monoid M is $\mathcal{R}$-trivial, this is equivalent to $h(\downarrow e) <_R h(\Downarrow e)$. We let $X_t = \{e \in E \mid h(\downarrow e) <_R h(\Downarrow e)\}$ be the set of all critical events of trace t.

▶ **Lemma 4.** *In the above notation, $|X_t| < |\mathcal{P}| \times |M|$.*

Proof. We show that the number of critical events on a single process line is less than $|M|$. This clearly implies that $|X_t|$ is less than $|\mathcal{P}| \times |M|$. For any $i \in \mathcal{P}$, the set of critical process i events is $X_t \cap E_i$. Assume $X_t \cap E_i = \{e_1, e_2, \cdots e_k\}$ with $e_1 < e_2 < \cdots < e_k$ (E_i is totally ordered by $\leq$). Since e_j are critical events we have $h(\downarrow e_j) <_R h(\Downarrow e_j)$. Since (the trace induced by) $\downarrow e_j$ is a prefix of (the trace induced by) $\Downarrow e_{j+1}$, we have $h(\Downarrow e_{j+1}) \leq_R h(\downarrow e_j)$. We get

$$h(\downarrow e_k) <_R h(\Downarrow e_k) \leq_R h(\downarrow e_{k-1}) <_R h(\Downarrow e_{k-1}) \leq_R \cdots h(\downarrow e_1) <_R h(\Downarrow e_1) \leq_R 1_M\,.$$

Therefore $k < |M|$ which concludes the proof. ◀

A configuration of t is a subset $C \subseteq E$ of events which is downward closed: $C = \downarrow C$. Clearly, for any event $e \in E$, $\downarrow e$ and $\Downarrow e$ are configurations of t. We sometimes abuse the notation and use C to also denote the trace induced by a configuration C of t. It is easy to see that $t = C \cdot t'$ where t' is the *trace* induced by the set $E \setminus C$ of events of t which do not belong to C.

A configuration C of t is *critical* if for every maximal event $e \in C$, we have $h(C) \neq h(C \setminus \{e\})$, equivalently, $h(C) <_R h(C \setminus \{e\})$. Note that if e is a maximal event of C then $C \setminus \{e\}$ is a configuration of t.

▶ **Lemma 5.** *Every maximal event of a critical configuration is a critical event.*

Proof. Suppose for contradiction that there exists a critical configuration C and a maximal event $e \in C$ such that $h(C) <_R h(C \setminus \{e\})$ but $h(\downarrow e) = h(\Downarrow e)$. We have that $h(C) = h(\downarrow e) \cdot h(C \setminus \downarrow e)$. Substituting $h(\downarrow e)$ by $h(\Downarrow e)$, we get $h(C) = h(\Downarrow e) \cdot h(C \setminus \downarrow e) = h(C \setminus \{e\})$ giving us a contradiction to the fact that $h(C) \neq h(C \setminus \{e\})$. ◀

▶ **Example 6.** A configuration in which every maximal event is critical, may not itself be critical. We show this via an example. Consider three processes and the distributed alphabet defined by $\Sigma_1 = \{a\}$, $\Sigma_2 = \{b\}$ and $\Sigma_3 = \{c\}$ (each process has a local action). Consider the $\mathcal{R}$-trivial (commutative) monoid $M = (\{0, 1, 2\}, \oplus, 0)$ with truncated sum defined as $x \oplus y = \min(2, x + y)$. In other words, 0 is the neutral element, 2 is absorbent, and $1 \oplus 1 = 2$. We have $2 <_R 1 <_R 0$. Consider the (truncated) length trace morphism $h\colon TR(\tilde{\Sigma}) \to M$ defined by $h(x) = 1$ for all $x \in \Sigma$. Finally, let $t = abc = (E, \leq, \lambda)$ be the trace with 3 events labelled a, b and c respectively. Each event e in t is critical and is maximal in t. But the configuration E is not critical since for each $e \in E$, we have $h(E \setminus \{e\}) = 2 = h(E)$.

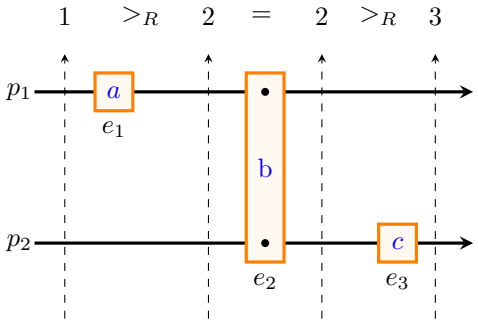
▶ **Lemma 7.** *Any prefix t' of t containing all the critical events of t, evaluates to the same monoid value as t under h. In other words, we have $h(\downarrow X_t) = h(t)$.*

Proof. We have $h(t) \leq_R h(\downarrow X_t)$ since $\downarrow X_t$ is a prefix of t. For the converse, we first show that there exists a *critical* configuration C of $t = (E, \leq, \lambda)$ such that $h(C) = h(t)$. The configuration $C = E$ satisfies $h(C) = h(t)$. Let C be a configuration with $h(C) = h(t)$.

Either C is critical and we are done, or we can drop a maximal event from C to get a strictly smaller configuration C' satisfying $h(C') = h(t)$. Repeating this, we find a critical configuration C of t such that $h(C) = h(t)$. By Lemma 5, the maximal events of C are critical. Hence we get $C \subseteq {\downarrow}X_t$. This implies $h({\downarrow}X_t) \leq_R h(C) = h(t)$. By $\mathcal{R}$-triviality, we deduce $h({\downarrow}X_t) = h(t)$. ◀

The *type* of a trace $t = (E, \leq, \lambda)$ is the Σ-labelled poset $\mathsf{type}(t) = (X_t, \leq, \lambda)$ obtained as the restriction of the labelled poset $(E, \leq, \lambda)$ to the critical events of t.

▶ **Example 8.** It is important to note that the type of a trace need not necessarily be a trace itself. Let $t = abc$ be a trace over the distributed alphabet $\Sigma_1 = \{a, b\}$, $\Sigma_2 = \{b, c\}$ with 3 events $e_1 < e_2 < e_3$ labelled a, b, c respectively, as shown in the figure below. Consider the $\mathcal{R}$-trivial (commutative) monoid $M = (\{1, 2, 3\}, \max, 1)$ where max is the binary maximum operation. It is clear that $3 <_R 2 <_R 1$. Consider the trace morphism $h\colon TR(\tilde{\Sigma}) \to M$ defined by $h(a) = 2$, $h(b) = 1$ and $h(c) = 3$. We see that only e_1 and e_3 are critical events of t. Then, $\mathsf{type}(t) = (\{e_1, e_3\}, \leq, \lambda)$ with $e_1 < e_3$ which is induced by the original partial order on t. Since the labels a and c are independent, $\mathsf{type}(t)$ is not a trace.



Let $s = (Z, \leq, \lambda)$ be a Σ-labelled poset. A linearization of s is a word[1] $w = (Z, \preceq, \lambda)$ where $\preceq$ is a total order on Z which is consistent with $\leq$, that is, satisfying $\leq \subseteq \preceq$. We denote by $\mathsf{Lin}(s) \subseteq \Sigma^*$ the set of linearizations of s.

Let $h'\colon \Sigma^* \to M$ be the *word* morphism defined by $h'(a) = h(a)$ for all $a \in \Sigma$. For all traces $t \in TR(\tilde{\Sigma})$ and for all linearizations $w \in \mathsf{Lin}(t)$, we have $h'(w) = h(t)$.

For every monoid element $m \in M$ we define $\Sigma^m = \{a \in \Sigma \mid m \cdot h(a) = m\}$ as the set of letters stabilizing m. The complement $\Sigma \setminus \Sigma^m = \{a \in \Sigma \mid m \cdot h(a) \neq m\}$ also denoted $\overline{\Sigma^m}$ is the set of letters which cause an $\mathcal{R}$-descent for m. This means that for every $a \in \overline{\Sigma^m}$ $m \cdot h(a) <_R m$.

▶ **Lemma 9.** *Let $t = (E, \leq, \lambda)$ be a trace and $\mathsf{type}(t) = (X_t, \leq, \lambda)$ be its type. For all linearizations $w \in \mathsf{Lin}(\mathsf{type}(t))$, we have $h'(w) = h(t)$.*

Proof. The proof is by induction on the number $k = |X_t|$ of critical events of t.

For the base case $k = 0$ there are no critical events in t. This means that $w = \varepsilon$ and all events in t have labels in Σ^{1_M}. Clearly, we have $h'(\varepsilon) = 1_M = h(t)$.

Assume now $k > 0$ and that the result holds for all traces s with $|X_s| < k$. Consider the linearization $w = a_1 a_2 \cdots a_k \in \mathsf{Lin}(\mathsf{type}(t))$. We have $w = (X_t, \preceq, \lambda)$ with $X_t = \{e_1, e_2, \ldots, e_k\}$, $e_1 \prec e_2 \prec \cdots \prec e_k$ and $\lambda(e_i) = a_i$. Note that the event e_k is maximal in

[1] A word $a_1 a_2 \cdots a_n \in \Sigma^*$ is identified with the structure $(\{e_1, e_2, \ldots, e_n\}, \preceq, \lambda)$ with $\lambda(e_i) = a_i$ and $e_1 \prec e_2 \prec \cdots \prec e_n$.

$\downarrow X_t$. Consider the trace s which is the prefix of t induced by $(\downarrow X_t) \setminus \{e_k\}$. It is easy to see that $\mathsf{type}(s) = (\{e_1, \ldots, e_{k-1}\}, \leq, \lambda)$ and $v = a_1 \cdots a_{k-1} \in \mathsf{Lin}(\mathsf{type}(s))$. By induction, we have $h'(v) = h(s)$. Therefore, $h'(w) = h'(v) \cdot h'(a_k) = h(s) \cdot h(\lambda(e_k)) = h(\downarrow X_t) = h(t)$ where the last equality follows from Lemma 7. ◀

It follows immediately that the type of a trace determines its value under h.

▶ **Corollary 10.** *Let $t, t' \in TR(\tilde{\Sigma})$ be two traces. If $\mathsf{type}(t)$ and $\mathsf{type}(t')$ are isomorphic, denoted $\mathsf{type}(t) \cong \mathsf{type}(t')$, then $h(t) = h(t')$.*

We conclude this subsection with some more definitions. Recall that we have fixed the distributed alphabet $\tilde{\Sigma}$ and the morphism h from $TR(\tilde{\Sigma})$ to the finite $\mathcal{R}$-trivial monoid M. We define the *type set* of $\tilde{\Sigma}$, with respect to h, to be the set of all possible types of all traces over $\tilde{\Sigma}$. More precisely, $\mathsf{types}(\tilde{\Sigma}) = \{\mathsf{type}(t) \mid t \in TR(\tilde{\Sigma})\}$. Note that, it clearly depends on h. By Lemma 4, every type is a Σ-labelled poset whose underlying set has size less than $|\mathcal{P}| \times |M|$. Therefore $\mathsf{types}(\tilde{\Sigma})$ is a finite set.

By Corollary 10, the type of a trace determines its monoid value under the recognizing morphism h. Hence we can define a function $\pi \colon \mathsf{types}(\tilde{\Sigma}) \to M$ by $\pi(\mathsf{type}(t)) = h(t)$ for all $t \in TR(\tilde{\Sigma})$. Note that, by Lemma 9, if $\sigma = (X, \leq, \lambda)$ is a type then $\pi(\sigma) = h'(w)$ for all $w \in \mathsf{Lin}(\sigma)$.

Let $t = (E, \leq, \lambda)$ be a trace and $e \in E$ be an event. The *type of e in t* is the type of the prefix of t induced by the *strict past* of e: $\mathsf{type}(t, e) = \mathsf{type}(\Downarrow e) = (X_{\Downarrow e}, \leq, \lambda) = (X_t \cap \Downarrow e, \leq, \lambda)$. We have $\pi(\mathsf{type}(t, e)) = h(\Downarrow e)$.

▶ **Lemma 11.** *Let $t = (E, \leq, \lambda)$ be a trace and e, f be events in t. If $\mathsf{type}(t, e) = \mathsf{type}(t, f) = \sigma$ and $\lambda(e) = \lambda(f) \in \overline{\Sigma^m}$ where $m = \pi(\sigma)$ then $e = f$.*

Proof. Assume towards a contradiction that there are two events e, f in t with $\mathsf{type}(t, e) = \mathsf{type}(t, f) = \sigma$ and $\lambda(e) = \lambda(f) = a \in \overline{\Sigma^m}$ where $m = \pi(\sigma)$. Without loss of generality we may assume $e < f$ since they are both labelled with the same letter and hence are ordered. Then, e is a critical event: indeed, $\mathsf{type}(t, e) = \sigma$ hence $h(\Downarrow e) = \pi(\sigma) = m$ and $\lambda(e) = a \in \overline{\Sigma^m}$ implies that $h(\downarrow e) = m \cdot h(a) \neq m$. We deduce that $X_{\Downarrow e} \cup \{e\} \subseteq X_{\Downarrow f}$: there are more critical events in the strict past of f than in the strict past of e. This is a contradiction with $\mathsf{type}(t, e) = \mathsf{type}(t, f)$. ◀

3.3 Proof of Theorem 2 [1 → 2]

Having done the necessary setup we now prove the first direction of Theorem 2 (restated as Theorem 14 at the end of this subsection). We start with Lemma 12 which supplies the key argument by constructing $\mathsf{LocTL}[\mathsf{YP}]$ event formulas for identifying the type of an event.

An event formula φ of $\mathsf{LocTL}[\mathsf{YP}]$ is called *a strict past* formula if it is a boolean combination of formulas of the form $\mathsf{YP}\,\varphi'$. A strict past formula does not test the label of the current event.

▶ **Lemma 12.** *For every $\sigma \in \mathsf{types}(\tilde{\Sigma})$, there exists a strict past event formula φ_σ such that, for all traces $t = (E, \leq, \lambda) \in TR(\tilde{\Sigma})$ and all events e in t, $t, e \models \varphi_\sigma$ iff $\mathsf{type}(t, e) \cong \sigma$.*

Proof. We construct φ_σ by induction and prove that it has the correct property.

Base case: $|\sigma| = 0$, i.e., $\sigma = (\emptyset, \emptyset, \emptyset)$. Note that $\pi((\emptyset, \emptyset, \emptyset)) = 1_M = h(\varepsilon)$ where ε is the empty trace. We define $\varphi_\sigma = \mathsf{YH}(\Sigma^{1_M})$. We now prove that it has the desired property.

If $t, e \models \mathsf{YH}(\Sigma^{1_M})$ then every letter in the strict past of e stabilizes the monoid identity element. We deduce easily that there are no critical events in the strict past of e. Therefore $\mathsf{type}(t, e) = (\emptyset, \emptyset, \emptyset)$.

Conversely, if $\mathsf{type}(t, e) = (\emptyset, \emptyset, \emptyset)$ then there are no critical events in the strict past of e. Since the empty trace is the smallest prefix of a trace and it maps to 1_M under h, we deduce easily that every letter in the strict past of e must stabilize the monoid identity 1_M. Therefore $t, e \models \mathsf{YH}(\Sigma^{1_M})$.

Induction hypothesis: Let $k \geq 0$ and assume that for every type σ with $|\sigma| \leq k$, we have constructed φ_σ satisfying the property of the lemma. Consider an arbitrary type $\tau \in \mathsf{types}(\tilde{\Sigma})$ of size $k+1$. We construct φ_τ below and prove its correctness.

Let $s = (E_s, \leq_s, \lambda_s) \in TR(\tilde{\Sigma})$ be a trace with $\tau = \mathsf{type}(s) = (X_s, \leq_s, \lambda_s)$. For an element $x \in X_s$, we let $\tau_x = (\Downarrow_\tau x, \leq_s, \lambda_s)$ be the restriction of the labelled poset τ to the critical events strictly below x ($\Downarrow_\tau x = \{y \in X_s \mid y <_s x\}$). It is easy to see that $\tau_x = \mathsf{type}(s, x) = \mathsf{type}(\Downarrow_s x)$ where $\Downarrow_s x = \{e \in E_s \mid e <_s x\}$ is (the prefix of s induced by) the strict past of x in s. Let $m_x = h(\Downarrow_s x) = \pi(\tau_x)$ and $a_x = \lambda_s(x)$. Since $x \in X_s$ is a critical event of s, we have $h(\downarrow_s x) = h(\Downarrow_s x) \cdot h(\lambda_s(x)) = m_x h(a_x) <_R m_x = h(\Downarrow_s x)$. Therefore, $a_x \in \overline{\Sigma^{m_x}} = \overline{\Sigma^{\pi(\tau_x)}}$.

Notice that the size of $\tau_x \in \mathsf{types}(\tilde{\Sigma})$ is at most k. By induction hypothesis, we have a formula φ_{τ_x} satisfying the property of the lemma. We use the macro $\varphi(\tau, x)$ to denote the event formula $\varphi_{\tau_x} \wedge a_x$. Finally, if $Z \subseteq X_s$, we let $\tau_Z = (\downarrow_\tau Z, \leq_s, \lambda_s)$. Then, $\tau_Z = \mathsf{type}(\downarrow_s Z)$ is a type and we let $m_Z = \pi(\tau_Z) = h(\downarrow_s Z)$. We define

$$\varphi_\tau = \varphi_1 \wedge \varphi_2 \wedge \varphi_{order} \qquad \varphi_1 = \mathsf{YH}(\varphi_{crit} \vee \varphi_{stab})$$

$$\varphi_2 = \bigwedge_{x \in X_s} \mathsf{YP}\, \varphi(\tau, x) \qquad \varphi_{crit} = \bigvee_{x \in X_s} \varphi(\tau, x)$$

$$\varphi_{order} = \bigwedge_{x,y \in X_s \mid x <_s y} \mathsf{YP}\left(\varphi(\tau, y) \wedge \mathsf{YP}\, \varphi(\tau, x)\right) \wedge \bigwedge_{x,y \in X_s \mid x \not<_s y} \neg\, \mathsf{YP}\left(\varphi(\tau, y) \wedge \mathsf{YP}\, \varphi(\tau, x)\right)$$

$$\varphi_{stab} = \bigvee_{Z = \downarrow_\tau Z \subseteq X_s} \Sigma^{m_Z} \wedge \bigwedge_{x \in Z} \mathsf{YP}\, \varphi(\tau, x) \wedge \bigwedge_{x \in X_s \setminus Z} \neg\, \mathsf{YP}\, \varphi(\tau, x)$$

Before proceeding to the formal proof of correctness of φ_τ, we first provide an intuitive explanation of the various parts of φ_τ.

- φ_2: asserts that for every "event" x in τ, there is a (unique) corresponding critical event in the strict past whose type is isomorphic to τ_x and whose label is same as that of x.
- φ_1: asserts that every event in the strict past either corresponds to some "event" x in τ (satisfies φ_{crit}) or is labelled with a stabilizing letter for the type Z arising out of the "events" x in τ
- φ_{order}: asserts that the ordering between "events" in τ and the corresponding critical events is identical.

We now formally show that φ_τ indeed satisfies the property of the lemma. Towards this, fix a trace $t = (E_t, \leq_t, \lambda_t)$ and an event e in t.

($\rightarrow$) Assume that $t, e \models \varphi_\tau$. We show that $\mathsf{type}(t, e) \cong \tau$.

First consider the fact that $t, e \models \varphi_2$. This means that for all $x \in X_s$, there exists an event $e_x \in \Downarrow_t e$, such that $t, e_x \models \varphi(\tau, x)$. By the inductive hypothesis, we have $\tau_x \cong \mathsf{type}(t, e_x) = \mathsf{type}(\Downarrow_t e_x)$ and $\lambda_t(e_x) = a_x \in \overline{\Sigma^{m_x}}$. From Lemma 11, e_x is the unique event in t satisfying $\varphi(\tau, x)$. Using $h(\Downarrow_t e_x) = \pi(\tau_x) = m_x$ and $\lambda_t(e_x) = a_x \in \overline{\Sigma^{m_x}}$, we deduce that e_x is critical in $\Downarrow_t e$. Therefore, $Y = \{e_x \mid x \in X_s\} \subseteq X_{\Downarrow_t e}$.

Now, using the fact that $t, e \models \varphi_{order}$ and the unicity of the events e_x, we deduce that for all $x, y \in X_s$, we have $x \leq_s y$ if and only if $e_x \leq_t e_y$. We also know that for all $x \in X_s$, we have $\lambda_s(x) = a_x = \lambda_t(e_x)$. Therefore, $\tau \cong (Y, \leq_t, \lambda_t)$.

In order to obtain $\tau \cong \mathsf{type}(t, e) = (X_{\Downarrow_t e}, \leq_t, \lambda_t)$ as desired, it remains to prove that $Y = X_{\Downarrow_t e}$. Towards a contradiction, assume that this is not the case and consider a minimal event $f \in X_{\Downarrow_t e} \setminus Y$. Since $t, e \models \varphi_1$, we have $t, f \models \varphi_{crit} \vee \varphi_{stab}$. If $t, f \models \varphi_{crit}$ then $t, f \models \varphi(\tau, x)$ for some $x \in X_s$ and by the unicity constraint $f = e_x \in Y$, a contradiction. Hence, $t, f \models \varphi_{stab}$ and there is a downward closed set $Z = \downarrow_\tau Z \subseteq X_s$ such that $\lambda_t(f) \in \Sigma^{m_Z}$ and $\Downarrow_t f \cap Y = \{e_x \mid x \in Z\}$. By minimality of f, we get $\Downarrow_t f \cap X_{\Downarrow_t e} = \Downarrow_t f \cap Y = \{e_x \mid x \in Z\}$. We deduce that $\mathsf{type}(t, f) = (\{e_x \mid x \in Z\}, \leq_t, \lambda_t) \cong \tau_Z$ and $h(\Downarrow_t f) = m_Z$. Using $\lambda_t(f) \in \Sigma^{m_Z}$ we conclude that f is not critical, a contradiction.

($\leftarrow$) Conversely, assume that $\tau \cong \mathsf{type}(t, e)$. We show that $t, e \models \varphi_\tau$.

From the isomorphism between $\tau = (X_s, \leq_s, \lambda_s)$ and $\mathsf{type}(t, e) = (X_{\Downarrow_t e}, \leq_t, \lambda_t)$ we get a one-one correspondence $x \in X_s \mapsto e_x \in X_{\Downarrow_t e}$ such that for all $x, y \in X_s$ we have $x \leq_s y$ if and only if $e_x \leq_t e_y$ and $\lambda_t(e_x) = \lambda_s(x)$. We deduce that $\mathsf{type}(t, e_x) \cong \tau_x$ and $\lambda_t(e_x) = a_x$ for all $x \in X_s$. Therefore, $t, e_x \models \varphi(\tau, x)$ for all $x \in X_s$ and $t, e \models \varphi_2$.

Next, it is easy to check that $t, e \models \varphi_{order}$ using the unicity property (Lemma 11) and the fact that $x <_s y$ if and only if $e_x <_t e_y$ for all $x, y \in X_s$.

Let $f \in \Downarrow_t e$ be an event in the strict past of e. Either $f \in X_{\Downarrow_t e}$ is a critical event and $f = e_x$ for some $x \in X_s$. In this case, we get $t, f \models \varphi(\tau, x)$ and $t, f \models \varphi_{crit}$. Or $f \in \Downarrow_t e \setminus X_{\Downarrow_t e}$ is a non-critical event. Let $Z = \{x \in X_s \mid e_x < f\}$. Then, $Z = \downarrow_\tau Z$ is downward closed and using the unicity property we deduce that $t, f \models \bigwedge_{x \in Z} \mathsf{YP}\, \varphi(\tau, x) \wedge \bigwedge_{x \in X_s \setminus Z} \neg\, \mathsf{YP}\, \varphi(\tau, x)$. Now, due to the fact that $\tau \cong \mathsf{type}(t, e)$ it is clear that $\tau_Z \cong \mathsf{type}(t, f)$. We deduce that $m_Z = \pi(\tau_Z) = h(\Downarrow_t f)$. Since f is not critical, we have $\lambda_t(f) \in \Sigma^{m_Z}$. Therefore, $t, f \models \varphi_{stab}$. We have shown that $t, e \models \varphi_1$. ◀

▶ **Lemma 13.** *For every element $m \in M$ we can define a strict past event formula φ_m in* LocTL[YP] *such that for every trace t and event e of t, we have $t, e \models \varphi_m$ iff $h(\Downarrow e) = m$.*

Proof. We define $\varphi_m = \bigvee_{\sigma \in \mathsf{types}(\tilde{\Sigma}) : \pi(\sigma) = m} \varphi_\sigma$ and show that it has the desired property.

Let t be a trace and e be an event in t. If $t, e \models \varphi_m$ then there exists some $\sigma \in \mathsf{types}(\tilde{\Sigma})$ such that $\pi(\sigma) = m$ and $t, e \models \varphi_\sigma$. By Lemma 12, $\sigma \cong \mathsf{type}(t, e) = \mathsf{type}(\Downarrow e)$. By definition of π, we have $h(\Downarrow e) = \pi(\sigma)$. As $\pi(\sigma) = m$, we conclude that $h(\Downarrow e) = m$. Now for the other direction assume $h(\Downarrow e) = m$. By Lemma 12, $t, e \models \varphi_{\mathsf{type}(t,e)}$ and by definition of π, $\pi(\mathsf{type}(t, e)) = h(\Downarrow e)$. As $h(\Downarrow e) = m$, $\pi(\mathsf{type}(t, e)) = m$ and hence $t, e \models \varphi_m$. ◀

▶ **Theorem 14.** *Let $L \subseteq TR(\tilde{\Sigma})$ be recognized by the morphism $h : TR(\tilde{\Sigma}) \to M$. Then there is a sentence Φ_L of* LocTL[YP] *which defines L: for all traces $t \in TR(\tilde{\Sigma})$, $t \models \Phi_L \iff t \in L$.*

Proof. As L is recognized by h, there is a subset $P \subseteq M$ such that $L = h^{-1}(P)$. For each monoid element $m \in M$, we define below a sentence Φ_m of LocTL[YP] which defines the trace language $h^{-1}(m)$: for all traces t we have $t \models \Phi_m \iff h(t) = m$. The sentence defining L is $\Phi_L = \bigvee_{m \in P} \Phi_m$.

Let $\# \notin \Sigma$ be a new letter. We extend the distributed alphabet $\tilde{\Sigma}$ by making $\#$ an action shared by all processes: $\tilde{\Sigma}_\# = \{\Sigma_i \cup \{\#\}\}_{i \in \mathcal{P}}$. For each trace $t \in TR(\tilde{\Sigma})$, we consider the augmented trace $\bar{t} = t \cdot \# \in TR(\tilde{\Sigma}_\#)$ obtained by adding a maximum event $e_\#$ labelled $\#$ to t. Note that all processes participate in $e_\#$ and $t = \Downarrow e_\#$. Now by Lemma 13, we have a

strict past event formula[2] φ_m, such that $\bar{t}, e_\# \models \varphi_m \iff h(\Downarrow e_\#) = m \iff h(t) = m$. We can now set Φ_m as the sentence obtained by replacing every outermost occurrence of YP in φ_m by E. It is easy to see that $t \models \Phi_m$ iff $\bar{t}, e_\# \models \varphi_m$. As a result, $t \models \Phi_m$ iff $h(t) = m$. ◀

4 Logic to cascade products to algebra

In this section, we first introduce a localized version of $\mathsf{LocTL}[\mathsf{YP}]$ namely $\mathsf{LocTL}[\mathsf{YP_i}]$ and show that both have the same expressive power. $\mathsf{LocTL}[\mathsf{YP_i}]$ is more suited for the direction $2 \to 3$ of Theorem 2 where we connect logic to cascade products of asynchronous automata. Subsequently, in Section 4.2 we introduce asynchronous devices and their cascade products and prove Theorem 22 in Section 4.3. Finally we show the $3 \to 1$ direction of Theorem 2 by connecting cascade products to algebra in Theorem 24.

4.1 $\mathsf{LocTL}[\mathsf{YP_i}]$ and its equivalence with $\mathsf{LocTL}[\mathsf{YP}]$

We define *local* modalities E_i and YP_i for each process $i \in \mathcal{P}$ as follows:

$$\mathsf{E}_i\,\varphi = \mathsf{E}(\Sigma_i \wedge \varphi) \quad \text{and} \quad \mathsf{YP}_i\,\varphi = \Sigma_i \wedge \mathsf{YP}(\Sigma_i \wedge \varphi)$$

Let t be a trace and e be an event of t. It is clear that, $t, e \models \mathsf{YP}_i\,\varphi$ iff e is an i-event and there exists an i-event e' in the strict past of e such that $t, e' \models \varphi$. Further $t \models \mathsf{E}_i\,\varphi$ iff there exists an i-event f of t such that $t, f \models \varphi$.

We define $\mathsf{LocTL}[\mathsf{YP_i}]$ as a fragment of $\mathsf{LocTL}[\mathsf{YP}]$ in which only above local modalities are allowed. More precisely, the syntax of $\mathsf{LocTL}[\mathsf{YP_i}]$ logic is as follows:

$$\Phi ::= \mathsf{E}_i\,\varphi \mid \Phi_1 \vee \Phi_2 \mid \neg\Phi$$
$$\varphi ::= a \mid \varphi_1 \vee \varphi_2 \mid \neg\varphi \mid \mathsf{YP}_i\,\varphi$$

▶ **Lemma 15.** $\mathsf{LocTL}[\mathsf{YP}]$ *and* $\mathsf{LocTL}[\mathsf{YP_i}]$ *express the same class of trace languages over* $\tilde{\Sigma}$.

Proof. We only need to show how to express the modalities E and YP of $\mathsf{LocTL}[\mathsf{YP}]$ with the *local* modalities of $\mathsf{LocTL}[\mathsf{YP_i}]$. We have $\mathsf{E}\,\varphi = \bigvee_{i\in\mathcal{P}} \mathsf{E}_i\,\varphi$.

For the strict past modality YP we use a well-known fact on traces. Let $t = (E, \leq, \lambda)$ be a trace and e, f be events in t. Then $e < f$ if and only if there is a sequence of events $e = e_0 < e_1 < e_2 < \cdots < e_k = f$ with $1 \leq k \leq |\mathcal{P}|$ and for each $1 \leq j \leq k$, there is a process $i_j \in \mathcal{P}$ such that e_{j-1}, e_j are both i_j-events. It is easy to deduce from here that

$$\mathsf{YP}\,\varphi = \bigvee_{\substack{i_1,i_2,\ldots,i_k\in\mathcal{P}\\ 1\leq k\leq|\mathcal{P}|}} \mathsf{YP}_{i_k} \cdots \mathsf{YP}_{i_2}\,\mathsf{YP}_{i_1}\,\varphi\,.$$

This shows that YP modality can be expressed in $\mathsf{LocTL}[\mathsf{YP_i}]$ and concludes the proof. ◀

▶ Remark 16. Referring to the syntax and semantics from [1] (Section 2.2), it is clear that $\mathsf{LocTL}[\mathsf{YP_i}]$ is a sublogic of LocPastPDL. Indeed, let φ be a $\mathsf{LocTL}[\mathsf{YP_i}]$ event formula and assume there exists an equivalent event formula $\overline{\varphi}$ in LocPastPDL. Then $\mathsf{YP}_i\,\varphi$ is implemented with $\langle \leftarrow_i^+ \rangle \overline{\varphi}$ and $\mathsf{E}_i\,\varphi$ is implemented with $\mathsf{EM}_i(\langle \leftarrow_i^* \rangle \overline{\varphi})$

[2] Formally the formula φ_m may use $\#$. We can take care of this by using $\neg\Sigma$ wherever it occurs in the formula.

4.2 Asynchronous devices and cascade product

We now recall the model of asynchronous automata due to Zielonka [23].

A deterministic *asynchronous automaton* A over $\tilde{\Sigma}$ (or simply over Σ) is a tuple $A = (\{S_i\}_{i\in\mathcal{P}}, \{\delta_a\}_{a\in\Sigma}, s_{\text{in}})$ where

- for each $i \in \mathcal{P}$, S_i is a finite non-empty set of local i-states.
- for each $a \in \Sigma$, $\delta_a \colon S_a \to S_a$ is a deterministic joint transition function where $S_a = \prod_{i\in\mathsf{loc}(a)} S_i$ is the set of a-states.
- with $S = \prod_{i\in\mathcal{P}} S_i$ as the set of all global states, $s_{\text{in}} \in S$ is a designated initial global state.

Let $s \in S$ be a global state. We can write s as (s_a, s_{-a}) where s_a is the projection of s onto $\mathsf{loc}(a)$ and s_{-a} is the projection onto $\mathcal{P} \setminus \mathsf{loc}(a)$. For any letter $a \in \Sigma$, we can extend the joint transition function $\delta_a \colon S_a \to S_a$ to a global transition function $\Delta_a \colon S \to S$ on global states as follows: $\Delta_a((s_a, s_{-a})) = (\delta_a(s_a), s_{-a})$. For a trace $t \in TR(\tilde{\Sigma})$, we let Δ_ε be the identity function on S, and define $\Delta_t \colon S \to S$ as the composition $\Delta_t = \Delta_a \circ \Delta_{t'}$ when $t = t'a$. Note that Δ_t is well defined, since for any pair (a, b) of independent letters, $\Delta_a \circ \Delta_b = \Delta_b \circ \Delta_a$. We denote by $A(t)$ the global state $\Delta_t(s_{\text{in}})$ reached when A runs on t.

At this point we define the transition monoid $M(A)$ associated with the asynchronous automaton A. Let $M(A)$ be the finite set of functions $\{\Delta_t \mid t \in TR(\tilde{\Sigma})\}$. This set under the usual function composition operation is a monoid with Δ_ε as it's identity element.

For any asynchronous automaton A, if we fix some $F \subseteq S$ to be the set of accepting global states then we say $L(A, F) = \{t \mid A(t) \in F\}$ is the *trace language accepted by* A.

We also use asynchronous automata as letter-to-letter asynchronous transducers as in [19, 2, 3] which compute a relabelling function. Let Γ be a finite non-empty set. We can define a natural extended distributed alphabet, $\widetilde{\Sigma \times \Gamma}$ by inheriting the distribution from $\tilde{\Sigma}$ that is, $\mathsf{loc}((a, \gamma)) = \mathsf{loc}(a)$. $TR(\tilde{\Sigma})$ and $TR(\widetilde{\Sigma \times \Gamma})$ are the set of traces over the original and extended distributed alphabets. A function $\theta \colon TR(\tilde{\Sigma}) \to TR(\widetilde{\Sigma \times \Gamma})$ is called a Γ-*labelling function* if, for every $t = (E, \leq, \lambda) \in TR(\tilde{\Sigma})$, $\theta(t) = (E, \leq, (\lambda, \mu)) \in TR(\widetilde{\Sigma \times \Gamma})$. A Γ-labelling function decorates each event e of the trace t with a label $\mu(e)$ from Γ.

An *asynchronous* Γ-*transducer* over $\tilde{\Sigma}$ (or simply Σ) is a tuple $\widehat{A} = (A, \{\mu_a\})$ where $A = (\{S_i\}, \{\delta_a\}, s_{\text{in}})$ is an asynchronous automaton and each μ_a ($a \in \Sigma$) is a map $\mu_a \colon S_a \to \Gamma$. We overload notation and use $\widehat{A} \colon TR(\tilde{\Sigma}) \to TR(\widetilde{\Sigma \times \Gamma})$ to also denote the Γ-labelling function computed by the transducer $\widehat{A}$. For $t = (E, \leq, \lambda) \in TR(\tilde{\Sigma})$, we define $\widehat{A}(t) = (E, \leq, (\lambda, \mu)) \in TR(\widetilde{\Sigma \times \Gamma})$ such that, for every $e \in E$ with $\lambda(e) = a$, $\mu(e) = \mu_a(s_a)$ where $s = A(\Downarrow e)$.

An asynchronous automaton (resp. transducer) with local state sets $\{S_i\}$ is said to be *localized at process* i if all local state sets S_j with $j \neq i$ are singletons. It is *localized* if it is localized at some process. For an automaton/transducer localized at i, all non-trivial transitions are on letters in which process i participates.

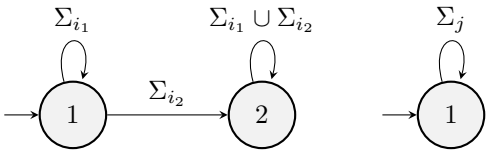


Figure 1 $U_1[i]$ automaton.

▶ **Example 17.** We define $U_1[i]$ to be a deterministic asynchronous automaton which is localized at process i. In $U_1[i]$, the local state set of process i has two states: $S_i = \{1, 2\}$ while all other processes have a single state. In Figure 1 we show the local automaton for

process i, which operates only on $\Sigma_i = \Sigma_{i_1} \uplus \Sigma_{i_2}$. Note that, a letter in Σ_{i_2} allows to change the local state from 1 to 2 while the rest of the transitions are self-loops. Clearly, we can identify the global-states of $U_1[i]$ with S_i. With this convention, we always use 1 as the initial global state for $U_1[i]$. Note that the transition monoid of $U_1[i]$ is isomorphic to a submonoid of U_1 from Example 1. For a trace t, $U_1[i](t)$, the global state reached by $U_1[i]$ when run on t, is 2 iff there exists an i-event in t with label in Σ_{i_2}.

A *degenerate* $U_1[i]$ automaton is a $U_1[i]$ automaton for which $\Sigma_{i_2} = \emptyset$. As state 2 is not reachable from the initial state 1, we can safely assume that $S_i = \{1\}$. Observe that the transition monoid of a degenerate $U_1[i]$ is the trivial monoid.

▶ **Example 18.** An asynchronous Γ-transducer over Σ whose underlying automaton is $U_1[i]$ from Example 17, is denoted by $\widehat{U_1[i]}$. More precisely, $\widehat{U_1[i]} = (U_1[i], \{\mu_a\})$. Recall that in $U_1[i]$, $S_i = \{1, 2\}$ and $S_j = \{1\}$ for $j \neq i$. Further, for each $a \in \Sigma$, $\mu_a \colon S_a \to \Gamma$. These output functions $\{\mu_a\}$ can be viewed as follows.

1. For each a such that $i \in \mathsf{loc}(a)$, we identify S_a with S_i. Thus, $\mu_a \colon S_i \to \Gamma$.
2. For each a such that $i \notin \mathsf{loc}(a)$, S_a is a singleton. As a result, μ_a is a constant function and can be simply thought of as an element of Γ.

Note that, $\widehat{U_1[i]}$ always uses 1 as the initial global state and it has an associated Γ-labelling function $\widehat{U_1[i]} \colon TR(\tilde{\Sigma}) \to TR(\widetilde{\Sigma \times \Gamma})$.

A *degenerate* $\widehat{U_1[i]}$ Γ-transducer has a degenerate $U_1[i]$ as its underlying automaton and has effectively only one global state. As a result, its output function can be simply specified as a output function $\mu \colon \Sigma \to \Gamma$.

Now we introduce the local cascade product of asynchronous transducers [4, 3] which will play an important role in later sections.

▶ **Definition 19.** *Let* $\widehat{A} = (\{S_i\}, \{\delta_a\}, s_{\text{in}}, \{\mu_a\})$ *be a* Γ*-transducer over* Σ *and* $\widehat{B} = (\{Q_i\}, \{\delta_{(a,\gamma)}\}, q_{\text{in}}, \{\nu_{(a,\gamma)}\})$ *be* Π*-transducer over* $\Sigma \times \Gamma$*. We define the local cascade product of* $\widehat{A}$ *and* $\widehat{B}$ *to be the* Π*-transducer* $\widehat{A} \circ_\ell \widehat{B} = (\{S_i \times Q_i\}, \{\nabla_a\}, (s_{\text{in}}, q_{\text{in}}), \{\tau_a\})$ *over* Σ *where* $\nabla_a((s_a, q_a)) = (\delta_a(s_a), \delta_{(a,\mu_a(s_a))}(q_a))$ *and* $\tau_a \colon S_a \times Q_a \to \Pi$ *is defined by* $\tau_a((s_a, q_a)) = \nu_{(a,\mu_a(s_a))}(q_a)$*.*

In the sequential case, that is, when $|\mathcal{P}| = 1$, the local cascade product coincides with the well-known operation of cascade product of sequential letter-to-letter transducers.

The following lemma is easily verified. See [3] for more details.

▶ **Lemma 20.** *The* Π*-labelling function computed by* $\widehat{A} \circ_\ell \widehat{B}$ *is the composition of the* Γ*-labelling function computed by* $\widehat{A}$ *and the* Π*-labelling function computed by* $\widehat{B}$*: for every* $t \in TR(\tilde{\Sigma})$*,*

$$(\widehat{A} \circ_\ell \widehat{B})(t) = \widehat{B}(\widehat{A}(t))$$

Although we have defined the local cascade product of transducers, it is equally easy to define the local cascade product of asynchronous automata. We refer to [3] for precise details of this construction and its connection with the wreath product operation and the resulting local cascade product principle.

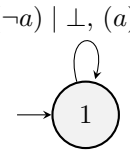
4.3 Proof of Theorem 2 [2 → 3]

In this section, we provide a proof of 2 → 3 of Theorem 2. In view of Lemma 15, it suffices to show that languages expressed by $\mathsf{LocTL}[\mathsf{YP_i}]$ sentences are accepted by local cascade product of $U_1[i]$ automata. This is precisely the content of Theorem 22.

Let φ be a $\mathsf{LocTL}[\mathsf{YP_i}]$ event formula. For each trace $t \in TR(\tilde{\Sigma})$ and event e in t, we let $\mu_\varphi(e) \in \{\top, \bot\}$ be the truth value of φ at e, and we let θ^φ be the $\{\top, \bot\}$-labelling function given by $\theta^\varphi(t) = (E, \leq, (\lambda, \mu_\varphi))$, for each $t = (E, \leq, \lambda) \in TR(\tilde{\Sigma})$.

▶ **Lemma 21.** *Let φ be a $\mathsf{LocTL}[\mathsf{YP_i}]$ event formula and let θ^φ be the corresponding $\{\top, \bot\}$-labelling function. Then we can construct a local cascade product $\widehat{A}_\varphi$ of $\widehat{U_1[i]}$ transducers which computes θ^φ.*

Proof. The construction of $\widehat{A}_\varphi$ proceeds by structural induction on φ. We first consider the base case with $\varphi = a$. Fix some process $i \in \mathsf{loc}(a)$. We let $\widehat{A}_\varphi$ be the degenerate $\{\top, \bot\}$-transducer $\widehat{U_1[i]}$ over Σ whose output function is $\mu\colon \Sigma \to \{\top, \bot\}$ where $\mu(b) = \top$ if $b = a$ and $\mu(b) = \bot$ otherwise. Clearly, the $\{\top, \bot\}$-labelling function computed by $\widehat{A}_\varphi$ coincides with θ^φ.

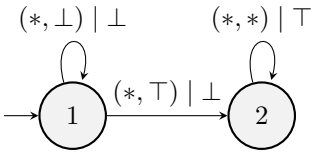


Now we consider the inductive case where $\varphi = \mathsf{YP}_i\, \varphi'$. By induction, we have $\{\top, \bot\}$-transducer $\widehat{A}_{\varphi'}$ which computes $\theta^{\varphi'}$. Note that the output alphabet of $\widehat{A}_{\varphi'}$ is $\Pi = \Sigma \times \{\top, \bot\}$ with the distribution $\Pi_i = \Sigma_i \times \{\top, \bot\}$.

We construct $\widehat{A}_\varphi$ as a local cascade product $\widehat{A}_{\varphi'} \circ_\ell \widehat{B}$, where $\widehat{B}$ is a $\{\top, \bot\}$-transducer $\widehat{U_1[i]}$ over Π defined below, so that $\widehat{A}_\varphi$ computes θ^φ. In view of Lemma 20, $\widehat{A}_\varphi(t) = \widehat{B}(\widehat{A}_{\varphi'}(t))$. By induction on φ', $\widehat{A}_{\varphi'} = \theta^{\varphi'}$. Thus, the requirement that $\widehat{A}_\varphi$ computes θ^φ simply translates to the requirement that $\widehat{B}(\theta^{\varphi'}(t)) = \theta^\varphi(t)$.

Let $t = (E, \leq, \lambda)$ be a trace over Σ and $t' = \theta^{\varphi'}(t) = (E, \leq, (\lambda, \mu'))$. Further, let $t'' = \theta^\varphi(t) = (E, \leq, (\lambda, \mu))$. As $\varphi = \mathsf{YP}_i\, \varphi'$, for an event $e \in E$, $\mu(e) = \top$ iff e is an i-event and there is an i-event f in the strict past of e such that $\mu'(f) = \top$.

We are now ready to construct $\widehat{B}$ as a $\widehat{U_1[i]}$ transducer over Π so as $\widehat{B}(t') = t''$. The local component for process i in $\widehat{B} = \widehat{U_1[i]}$ is as depicted in the figure below.



More precisely, for $\widehat{B}$ over Π, with $S_i = \{1, 2\}$, the only non-self-loop transitions (from local state 1 to local state 2) are on letters of the form $(a, \top)$ with $a \in \Sigma_i$. The output functions are as follows. For $a \in \Sigma_i$, $\mu_{(a,*)}(1) = \bot$, $\mu_{(a,*)}(2) = \top$ and for $a \notin \Sigma_i$, $\mu_{(a,*)} = \bot$.

Clearly, $\widehat{B}$ outputs $\bot$ on events in which process i does not participate. Further, it outputs $\top$ on an i-event only after reaching state 2 which ensures that there is an i-event in the strict past where the input label (the truth value of φ') carries the value $\top$. It is easy to see from here that, for every $t \in TR(\tilde{\Sigma})$, $\widehat{B}(\theta^{\varphi'}(t)) = \theta^\varphi(t)$. This in turn implies that $\widehat{A}_\varphi = \widehat{A}_{\varphi'} \circ_\ell \widehat{B}$ correctly computes θ^φ.

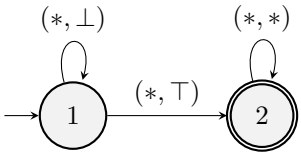
We now consider the case where $\varphi = \varphi_1 \vee \varphi_2$. By induction, we have a transducer $\widehat{A}_{\varphi_1}$ ($\widehat{A}_{\varphi_2}$) which computes θ^{φ_1} (respectively θ^{φ_2}). We consider the natural direct product $\widehat{A} = \widehat{A}_{\varphi_1} \times \widehat{A}_{\varphi_2}$ which is a $(\{\top, \bot\} \times \{\top, \bot\})$-transducer which simultaneously computes the truth values of φ_1 as well as φ_2. We can easily see that the direct product is a special

case of the cascade product. We can simply define $\widehat{A}_\varphi = \widehat{A} \circ_\ell \widehat{B}$ where $\widehat{B}$ is a degenerate $\{\top,\bot\}$-transducer $\widehat{U_1[i]}$ (for any process i) over $\Sigma \times (\{\top,\bot\} \times \{\top,\bot\})$ whose output function μ is as follows: $\mu((a,b_1,b_2)) = b_1 \vee b_2$. It is easy to see that $\widehat{A}$ computes θ^φ.

The remaining case where $\varphi = \neg\varphi'$ can be similarly handled by a local cascade product of $\widehat{A}_{\varphi'}$ with a degenerate $\widehat{U_1[i]}$ transducer. ◀

▶ **Theorem 22.** *Let Φ be a $\mathsf{LocTL}[\mathsf{YP_i}]$ sentence. There exists an asynchronous automaton A_Φ which is a cascade product of $U_1[i]$ automata and it accepts exactly the trace language defined by Φ. In other words, A_Φ accepts $L_\Phi = \{t \in TR(\tilde{\Sigma}) \mid t \models \Phi\}$.*

Proof. A basic $\mathsf{LocTL}[\mathsf{YP_i}]$ trace formula Φ is of the form $\mathsf{E}_j\,\varphi$, where φ is a $\mathsf{LocTL}[\mathsf{YP_i}]$ event formula. For a trace t, we have that $t \models \mathsf{E}_j\,\varphi$ iff there exists a j-event e in the trace such that $t, e \models \varphi$. From Lemma 21 we have a $\{\top,\bot\}$-transducer $\widehat{A}_\varphi$ which is a local cascade product of $\widehat{U_1[i]}$ transducers such that $\widehat{A}_\varphi = \theta^\varphi$. Let $t = (E, \leq, \lambda)$ be any trace over Σ. It is clear that $t \models \mathsf{E}_j\,\varphi$ iff some j-event in the output trace $t' = \widehat{A}_\varphi(t)$ is labelled by $\top$. Therefore the desired property of A_Φ is that it should accept the trace t iff t' has a $\top$-labelled j-event. To achieve this, we simply take the cascade product of the $\{\top,\bot\}$-transducer $\widehat{A}_\varphi$ with an asynchronous automaton $U_1[j]$ over $\Sigma \times \{\top,\bot\}$. Beginning in the initial global state 1, the non-trivial component of $U_1[j]$ automaton owned by process j waits at local state 1 for the label $\top$ to occur on j-events and transits to local state 2 after seeing the label $\top$ and stays there thereafter. This $U_1[j]$ automaton is depicted in the figure below.



In order to accept L_Φ, in the automaton $A_\Phi = \widehat{A}_\varphi \circ_\ell U_1[j]$, we define a global state of A_Φ to be *accepting* if its component for the last (in the cascade product) $U_1[j]$ global state is 2. As $\widehat{A}_\varphi$ is a local cascade product of $\widehat{U_1[i]}$ transducers and the underlying automaton for each $\widehat{U_1[i]}$ is simply a $U_1[i]$ automaton, it is not difficult to see that A_Φ is in fact a local cascade product of $U_1[i]$ automata.

This takes care of basic primitive languages defined by sentences of the form $\mathsf{E}_i\,\varphi$. As a general sentence Φ is a boolean combination of sentences of the form $\mathsf{E}_i\,\varphi$, L_Φ is also a boolean combination of basic primitive languages. The result for general Φ follows as boolean combinations can be handled by direct product construction which can be easily implemented by the local cascade product construction. ◀

4.4 Proof of Theorem 2 [3 → 1]

In this subsection, we provide a proof of 3 → 1 of Theorem 2 in the form of Theorem 24.

▶ **Lemma 23.** *Let A be an asynchronous automaton over Σ which is a local cascade product of $U_1[i]$ automata. Then the transition monoid of A is $\mathcal{R}$-trivial.*

Proof. Let $A = (\{S_i\}, \{\delta_a\}, s_{in})$ be a local cascade product of $U_1[i]$ automata. We show that there is a partial order $\leq$ on the global state set $S = \prod_{i\in\mathcal{P}} S_i$ such that, for every $(s,a) \in S \times \Sigma$, $\Delta_a(s) \leq s$ (recall that, $\Delta_a \colon S \to S$ is the natural global transition function induced by the joint transition function $\delta_a \colon S_a \to S_a$). It is known [6] and easy to see that the existence of such a partial order implies that the transition monoid of A is $\mathcal{R}$-trivial.

We show the existence of the required partial order by induction on the number of constituent $U_1[i]$ automata in A. In the base case, A itself is a $U_1[i]$. As the global state set of $U_1[i]$ is $\{1,2\}$ (see Example 17), by setting $\leq$ to be the total order with $2 < 1$, we get the desired property.

To complete the inductive step, we assume the existence of a partial order $\preceq$ on the global state set S of A. We now use it to construct the required partial order $\leq$ on the global state set of $A \circ_\ell U_1[i]$. Note that the global state set of $A \circ_\ell U_1[i]$ is simply $S \times \{1,2\}$. We define $\leq$ as follows: for $s, s' \in S$ and $k, k' \in \{1,2\}$, $(s',k') \leq (s,k)$ iff $s' \preceq s$, and $k' \leq k$. In other words, $\leq$ is simply the direct product of $\preceq$ on S and the total order on $\{1,2\}$ with $2 < 1$. Let $a \in \Sigma$, then the (global) deterministic transition on a of $A \circ_\ell U_1[i]$ takes the state (s,k) to the state $(s',k') = (\Delta_a(s), \Delta_{(a,s_a)}(k))$. It is clear from this description that $(s',k') \leq (s,k)$. This shows that $\leq$ on $S \times \{1,2\}$ has the desired property and we are done. ◀

▶ **Theorem 24.** *Let $L \subseteq TR(\tilde{\Sigma})$ be accepted by an asynchronous automaton A which is a local cascade product of $U_1[i]$ automata. Then L is $\mathcal{R}$-trivial.*

Proof. We consider the transition monoid $M(A)$ associated with A. Recall that it is the finite monoid consisting of $\{\Delta_t : t \in TR(\tilde{\Sigma})\}$ where Δ_t is the global transition function on the global state set induced by the trace t. By the previous lemma, this monoid is $\mathcal{R}$-trivial. Due to the local nature of the transitions of A, if a and b are independent then Δ_a and Δ_b commute. As a result, we have a well defined morphism $h : TR(\tilde{\Sigma}) \to M(A)$ which sends t to Δ_t. Suppose that L is accepted by A with s_{in} as the initial global state and F as a subset of global accepting states of A. Then L is recognized by the trace morphism h via the set $\{\Delta_t \in M(A) \mid \Delta_t(s_{in}) \in F\}$. Hence L is $\mathcal{R}$-trivial. ◀

5 Results on related fragments

In this section we briefly present other results which are of the same form as Theorem 2 but for trace languages which are $\mathcal{L}$-trivial or recognized by monoids from the class DA. First we shall define the necessary terminology and then subsequently state the results.

$\mathsf{LocTL}[\mathsf{XF}, \mathsf{YP}]$ is the fragment of temporal logic over traces which contains both the strict future (XF) and the strict past (YP) modality. We give below the syntax for $\mathsf{LocTL}[\mathsf{XF}, \mathsf{YP}]$ and only explain the semantics of the new modality XF.

$$\Phi ::= \mathsf{E}\,\varphi \mid \Phi_1 \vee \Phi_2 \mid \neg\Phi$$
$$\varphi ::= a \mid \varphi_1 \vee \varphi_2 \mid \neg\varphi \mid \mathsf{XF}\,\varphi \mid \mathsf{YP}\,\varphi$$

For a trace $t = (E, \leq, \lambda)$ and an event $e \in E$, we have $t, e \models \mathsf{XF}\,\varphi$ if there exists an event f in t with $e < f$ and $t, f \models \varphi$. Observe that $\mathsf{LocTL}[\mathsf{YP}]$ is the past fragment of $\mathsf{LocTL}[\mathsf{XF}, \mathsf{YP}]$. We can similarly define $\mathsf{LocTL}[\mathsf{XF}]$ is as the future fragment of $\mathsf{LocTL}[\mathsf{XF}, \mathsf{YP}]$ which has access to only XF modality.

For a trace $t = (E, \leq, \lambda)$, the reverse trace $t^r = (E, \leq', \lambda)$ is defined to be the trace where, for $e, f \in E$, $e \leq' f$ iff $f \leq e$. For a trace language $L \subseteq TR(\tilde{\Sigma})$, the reverse language L^r is defined as $L^r = \{t^r \mid t \in L\}$.

For a monoid M, M^r denotes the opposite monoid with multiplication $\circ^r$ defined as $m \circ^r n := n \circ m$ for all $m, n \in M^r$. It is easy to see that L is recognized by M iff L^r is recognized by M^r. Further, it it easy to deduce that M is $\mathcal{L}$-trivial iff M^r is $\mathcal{R}$-trivial. It follows that L is $\mathcal{L}$-trivial iff L^r is $\mathcal{R}$-trivial.

It is also easy to verify that L is $\mathsf{LocTL}[\mathsf{XF}]$ definable iff L^r is $\mathsf{LocTL}[\mathsf{YP}]$ definable. Given a sentence Φ in $\mathsf{LocTL}[\mathsf{XF}]$ defining L, we can obtain a $\mathsf{LocTL}[\mathsf{YP}]$ sentence Φ^r which defines L^r by simply replacing all occurrences of XF operators in Φ by YP, and vice versa. In view of the above discussion, it follows from Theorem 2 that a trace language L is $\mathcal{L}$-trivial iff it is definable in $\mathsf{LocTL}[\mathsf{XF}]$.

In order to obtain a translation of $\mathsf{LocTL}[\mathsf{XF}]$, or more generally, $\mathsf{LocTL}[\mathsf{XF},\mathsf{YP}]$ event formulas and sentences into cascade products of asynchronous devices, as in Lemma 21 and Theorem 22 for $\mathsf{LocTL}[\mathsf{YP}]$, we work with *localized* modalities XF_i and YP_i and use *non-deterministic* asynchronous automata/transducers.

We can define the localized logic $\mathsf{LocTL}[\mathsf{XF}_\mathsf{i},\mathsf{YP}_\mathsf{i}]$ by using the process indexed modalities XF_i, YP_i and E_i instead of XF, YP and E as in Section 4.1. The semantics of XF_i modality is as follows: for a trace $t = (E, \leq, \lambda)$ and $e \in E$, $t, e \models \mathsf{XF}_i\,\varphi$ iff e is an i-event and there exists an i-event f with $e < f$ such that $t, f \models \varphi$. It can be shown that $\mathsf{LocTL}[\mathsf{XF}_\mathsf{i},\mathsf{YP}_\mathsf{i}]$ has the same expressive power as $\mathsf{LocTL}[\mathsf{XF},\mathsf{YP}]$. Observe that $\mathsf{LocTL}[\mathsf{YP}_\mathsf{i}]$ is a fragment of $\mathsf{LocTL}[\mathsf{XF}_\mathsf{i},\mathsf{YP}_\mathsf{i}]$. One can similarly define the fragment $\mathsf{LocTL}[\mathsf{XF}_\mathsf{i}]$ of $\mathsf{LocTL}[\mathsf{XF}_\mathsf{i},\mathsf{YP}_\mathsf{i}]$ which has the same expressive power as $\mathsf{LocTL}[\mathsf{XF}]$.

Now we turn our attention to local cascade product of non-deterministic asynchronous automata and transducers. For the purpose of this work, it suffices to work with deterministic and *reverse* deterministic versions and their cascade products.

We begin with the key example of the reverse deterministic asynchronous automata $U_1^r[i]$. In $U_1^r[i]$, the local state set of process i has two states, that is, $S_i = \{1, 2\}$ while all other processes have a singleton state set $S_j = \{1\}$ for $j \neq i$. For a letter $a \notin \Sigma_i$, the processes participating in a have a deterministic transition, namely, a self-loop.

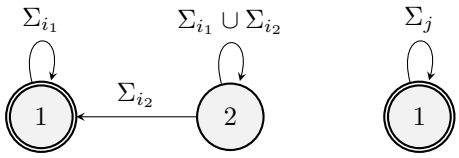


Figure 2 $U_1^r[i]$ automata: on the left is the local automaton for process i, on the right is the local single state automaton for every other process $j \neq i$.

If $a \in \Sigma_i = \Sigma_{i_1} \uplus \Sigma_{i_2}$, we can identify a joint a-state with the local state of process i. With this identification, process i has non-deterministic transitions as shown in Figure 2. For instance, assume $a \in \Sigma_{i_2}$. Then there is no transition on a at state 1. Further, at state 2, on a, the next state can be either state 1 or state 2. Note that these local transitions are obtained by simply *reversing* the local transitions in the *deterministic* $U_1[i]$ automata.

We *always* use the global state 1 as the only *final/accepting state*. It is important to observe that, for a trace t, $U_1^r[i]$ admits a *unique accepting run* ρ_t (which ends in global state 1). If t has no i-event with label in Σ_{i_2}, then ρ_t must begin in global state 1 and stays there until the end. However, if t has an i-event with label in Σ_{i_2}, then ρ_t must begin in the global state 2 and process i component changes its local state from 2 to 1 on the *last* occurrence of an i-event with label in Σ_{i_2}. After this crucial event, process i stays at local state 1 as all subsequent "future" i-events are labelled by letters in Σ_{i_1}. This unambiguity property of $U_1^r[i]$ is crucial. While using $U_1^r[i]$ as a language acceptor, we allow a *set* of initial global states. Thanks to the unambiguity property, the language accepted with single initial global state 1 is the complement of the language accepted with single initial global state 2.

Now we introduce the reverse deterministic asynchronous Γ-transducers $\widehat{U_1^r[i]}$. The underlying automaton of $\widehat{U_1^r[i]}$ is $U_1^r[i]$ and the output function provides a label from Γ on each local transition of $U_1^r[i]$. In $\widehat{U_1^r[i]}$, we allow any initial global state but insist

that 1 is the only accepting state. We also associate with $\widehat{U_1^r[i]}$ a Γ-labelling *function* $\widehat{U_1^r[i]}\colon TR(\tilde{\Sigma}) \to TR(\widetilde{\Sigma \times \Gamma})$ as follows: let $t = (E, \leq, \lambda)$ and ρ_t be the unique accepting run of the underlying $U_1^r[i]$ on t. We define $\widehat{U_1^r[i]}(t)$ to be the trace $t' = (E, \leq, (\lambda, \mu))$ where, for $e \in E$, $\mu(e)$ is the label from Γ on the transition used to "read" e on the unique run ρ_t.

Let $\Gamma = \{\top, \bot\}$. We now construct an instance of a Γ-transducer $\widehat{U_1^r[i]}$ over $\Sigma \times \Gamma$. This transducer, called $\widehat{U_{\mathsf{XF}_i}}$ below, implements the localized modality XF_i.

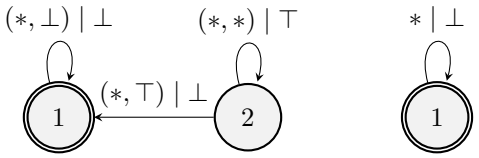


Figure 3 $\widehat{U_{\mathsf{XF}_i}}$ transducer which implements XF_i modality.

Consider a $\mathsf{LocTL}[\mathsf{XF_i}, \mathsf{YP_i}]$ formula $\varphi = \mathsf{XF}_i\, \varphi'$. We have the Γ-relabelling functions $\theta^{\varphi'}$ and θ^{φ} which keep track of truth values of φ' and φ respectively. Let $t = (E, \leq, \lambda)$ be a trace over Σ and $\theta^{\varphi'}(t) = (E, \leq, (\lambda, \mu'))$ and $\theta^{\varphi}(t) = (E, \leq, (\lambda, \mu))$ be traces over $\Sigma \times \Gamma$. For any event e of t, $\mu'(e) = \top$ iff $t, e \models \varphi'$ and $\mu(e) = \top$ iff $t, e \models \varphi$. Fix an event $e \in E$. By semantics of XF_i, $\mu(e) = \top$ iff e is an i-event and there exists a *strict future* i-event f (that is, $e < f$) such that $\mu'(f) = \top$. For the Γ-transducer $\widehat{U_{\mathsf{XF}_i}}$ over $\Sigma \times \Gamma$, depicted in Figure 3, it is easy to verify that, for every trace t over Σ, $\widehat{U_{\mathsf{XF}_i}}((\theta^{\varphi'}(t)) = \theta^{\varphi}(t)$.

In view of the above discussion, it is not surprising that the following theorem holds. We skip the technical details.

▶ **Theorem 25.** *Let $L \subseteq TR(\tilde{\Sigma})$. Then the following are equivalent.*

1. *L can be expressed by a sentence in $\mathsf{LocTL}[\mathsf{XF}, \mathsf{YP}]$.*
2. *L can be accepted by a cascade product of $U_1[i]$ and $U_1^r[i]$ automata.*

Even over words, the above result appears to be new. Further, we note that it has been shown in [17] (also see [11], [10]) that, over trace languages, $\mathsf{LocTL}[\mathsf{XF}, \mathsf{YP}]$ definability coincides with recognizability by monoids from the class DA.

The following theorem is a dual of our main theorem.

▶ **Theorem 26.** *Let $L \subseteq TR(\tilde{\Sigma})$. The following are equivalent.*

1. *L is a $\mathcal{L}$-trivial trace language.*
2. *L can be expressed by a sentence in $\mathsf{LocTL}[\mathsf{XF}]$.*
3. *L can be accepted by a cascade product of $U_1^r[i]$ automata.*

Recall that a trace language is $\mathcal{J}$-trivial iff it is $\mathcal{R}$-trivial as well as $\mathcal{L}$-trivial. This observation leads to the following result.

▶ **Theorem 27.** *Let $L \subseteq TR(\tilde{\Sigma})$. The following are equivalent.*

1. *L is a $\mathcal{J}$-trivial trace language.*
2. *L can be expressed by a sentence in $\mathsf{LocTL}[\mathsf{XF}]$ as well as a sentence in $\mathsf{LocTL}[\mathsf{YP}]$.*
3. *L can be accepted by a cascade product of $U_1^r[i]$ automata as well as another cascade product of $U_1[i]$ automata.*

References

1 Bharat Adsul, Paul Gastin, Shantanu Kulkarni, and Pascal Weil. An expressively complete local past propositional dynamic logic over Mazurkiewicz traces and its applications. In *Proceedings of the 39th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS'24, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3661814.3662110.

2 Bharat Adsul, Paul Gastin, Saptarshi Sarkar, and Pascal Weil. Wreath/cascade products and related decomposition results for the concurrent setting of Mazurkiewicz traces. In Igor Konnov and Laura Kovács, editors, *31st International Conference on Concurrency Theory, CONCUR 2020, September 1-4, 2020, Vienna, Austria (Virtual Conference)*, volume 171 of *LIPIcs*, pages 19:1–19:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPICS.CONCUR.2020.19.

3 Bharat Adsul, Paul Gastin, Saptarshi Sarkar, and Pascal Weil. Asynchronous wreath product and cascade decompositions for concurrent behaviours. *Log. Methods Comput. Sci.*, 18, 2022. doi:10.46298/LMCS-18(2:22)2022.

4 Bharat Adsul and Milind A. Sohoni. Asynchronous automata-theoretic characterization of aperiodic trace languages. In Kamal Lodaya and Meena Mahajan, editors, *FSTTCS 2004: Foundations of Software Technology and Theoretical Computer Science, 24th International Conference, Chennai, India, December 16-18, 2004, Proceedings*, volume 3328 of *Lecture Notes in Computer Science*, pages 84–96. Springer, 2004. doi:10.1007/978-3-540-30538-5_8.

5 Mikołaj Bojańczyk. Languages recognised by finite semigroups, and their generalisations to objects such as trees and graphs, with an emphasis on definability in monadic second-order logic, 2020. arXiv:2008.11635.

6 J.A. Brzozowski and Faith E. Fich. Languages of R-trivial monoids. *Journal of Computer and System Sciences*, 20(1):32–49, 1980. doi:10.1016/0022-0000(80)90003-3.

7 Joëlle Cohen, Dominique Perrin, and Jean-Eric Pin. On the expressive power of temporal logic. *Journal of Computer and System Sciences*, 46(3):271–294, 1993. doi:10.1016/0022-0000(93)90005-H.

8 Volker Diekert and Paul Gastin. Pure future local temporal logics are expressively complete for Mazurkiewicz traces. *Information and Computation*, 204(11):1597–1619, November 2006. doi:10.1016/J.IC.2006.07.002.

9 Volker Diekert, Paul Gastin, and Manfred Kufleitner. A survey on small fragments of first-order logic over finite words. *Int. J. Found. Comput. Sci.*, 19(3):513–548, 2008. doi:10.1142/S0129054108005802.

10 Volker Diekert, Martin Horsch, and Manfred Kufleitner. On first-order fragments for Mazurkiewicz traces. *Fundamenta Informaticae*, 80(1-3):1–29, 2007. URL: http://content.iospress.com/articles/fundamenta-informaticae/fi80-1-3-02.

11 Volker Diekert and Manfred Kufleitner. On first-order fragments for words and Mazurkiewicz traces: a survey. In *Proceedings of the 11th international conference on Developments in language theory*, pages 1–19, 2007.

12 Volker Diekert and Grzegorz Rozenberg, editors. *The Book of Traces*. World Scientific, 1995. doi:10.1142/2563.

13 Werner Ebinger and Anca Muscholl. Logical definability on infinite traces. *Theor. Comput. Sci.*, 154(1):67–84, 1996. doi:10.1016/0304-3975(95)00130-1.

14 Kousha Etessami and Thomas Wilke. An until hierarchy and other applications of an Ehrenfeucht-Fraïsse game for temporal logic. *Information and Computation*, 160(1):88–108, 2000. doi:10.1006/inco.1999.2846.

15 Giovanna Guaiana, Antonio Restivo, and Sergio Salemi. On aperiodic trace languages. In Christian Choffrut and Matthias Jantzen, editors, *STACS 91, 8th Annual Symposium on Theoretical Aspects of Computer Science, Hamburg, Germany, February 14-16, 1991, Proceedings*, volume 480 of *Lecture Notes in Computer Science*, pages 76–88. Springer, 1991. doi:10.1007/BFB0020789.

16 Johan Anthony Willem Kamp. *Tense Logic and the Theory of Linear Order*. University of California, Los Angeles, CA, USA, 1968.

17 Manfred Kufleitner. Polynomials, fragments of temporal logic and the variety DA over traces. *Theoretical Computer Science*, 376:89–100, 2007. Special issue DLT 2006. doi:10.1016/J.TCS.2007.01.014.

18 Robert McNaughton and Seymour A Papert. *Counter-Free Automata (MIT research monograph no. 65)*. The MIT Press, 1971.

19 Madhavan Mukund and Milind A. Sohoni. Keeping track of the latest gossip in a distributed system. *Distributed Comput.*, 10(3):137–148, 1997. doi:10.1007/s004460050031.

20 M.P. Schützenberger. On finite monoids having only trivial subgroups. *Information and Control*, 8(2):190–194, 1965. doi:10.1016/S0019-9958(65)90108-7.

21 Howard Straubing. *Finite automata, formal logic, and circuit complexity*. Birkhaüser Verlag, Basel, Switzerland, 1994.

22 Wolfgang Thomas. On logical definability of trace languages. In V. Diekert, editor, *Proceedings of a workshop of the ESPRIT Basic Research Action No 3166: Algebraic and Syntactic Methods in Computer Science (ASMICS), Kochel am See, Bavaria, FRG (1989)*, Report TUM-I9002, Technical University of Munich, pages 172–182, 1990.

23 Wieslaw Zielonka. Notes on finite asynchronous automata. *RAIRO Theor. Informatics Appl.*, 21(2):99–135, 1987. doi:10.1051/ita/1987210200991.

Omega-Regular Verification and Control for Distributional Specifications in MDPs

S. Akshay
Dept of CSE, Indian Institute of Technology Bombay, Mumbai, India

Ouldouz Neysari
Singapore Management University, Singapore
University of Tehran, Iran

Đorđe Žikelić
Singapore Management University, Singapore

Abstract

A classical approach to studying Markov decision processes (MDPs) is to view them as state transformers. However, MDPs can also be viewed as distribution transformers, where an MDP under a strategy generates a sequence of probability distributions over MDP states. This view arises in several applications, even as the probabilistic model checking problem becomes much harder compared to the classical state transformer counterpart. It is known that even distributional reachability and safety problems become computationally intractable (Skolem- and positivity-hard). To address this challenge, recent works focused on sound but possibly incomplete methods for verification and control of MDPs under the distributional view. However, existing automated methods are applicable only to distributional reachability, safety and reach-avoidance specifications.

In this work, we present the first automated method for verification and control of MDPs with respect to distributional omega-regular specifications. To achieve this, we propose a novel notion of distributional certificates, which are sound and complete proof rules for proving that an MDP under a distributionally memoryless strategy satisfies some distributional omega-regular specification. We then use our distributional certificates to design the first fully automated algorithms for verification and control of MDPs with respect to distributional omega-regular specifications. Our algorithms follow a template-based synthesis approach and provide soundness and relative completeness guarantees, while running in PSPACE. Our prototype implementation demonstrates practical applicability of our algorithms to challenging examples collected from the literature.

2012 ACM Subject Classification Theory of computation → Verification by model checking

Keywords and phrases MDPs, Distributional objectives, ω-regularity, Certificates

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2025.6

Funding This work was supported in part by the Singapore Ministry of Education (MOE) Academic Research Fund (AcRF) Tier 1 grant (Project ID:22-SIS-SMU-100), Google Research Award 2023 and the SBI Foundation Hub for Data Science & Analytics, IIT Bombay.

1 Introduction

Markov decision processes (MDPs) are a standard model for reasoning and sequential decision making in the presence of uncertainty. The verification community has long studied MDPs as state transformers, where their semantics are interpreted over cylinder sets of paths (see e.g. [8]). As a result, quantitative verification questions focus on state-based properties, such as the eventual reachability of a state with maximum probability over all MDP strategies. There is a rich body of literature on efficient algorithms for reasoning about state-based properties in MDPs, including model checking over expressive logics such as PCTL* [30].

An orthogonal class of objectives, which forms our focus in this paper, considers properties that are defined over the space of probability distributions over MDP states, rather than the

© S. Akshay, Ouldouz Neysari, and Đorđe Žikelić;
licensed under Creative Commons License CC-BY 4.0

state space of the MDP. This allows one to reason about the movement of the probability mass, for instance, one can say that always in the future the probability mass is equally divided between two bi-stable states. Such objectives, that we call *distributional objectives*, are simpler to reason about under alternative semantics which view MDPs as *distribution transformers*. In this view, starting from some initial distribution over MDP states, the MDP under a strategy induces a sequence of distributions over MDP states, generating a new distribution at each time step. One can then specify properties with respect to this sequence of distributions, such as distributional reachability or safety. This view naturally arises in several applications, including multi-agent systems [9, 5] or biochemical reaction networks [29, 28]. However, it turns out that even the simplest distributional properties such as distributional reachability and safety cannot be expressed in PCTL* [10], rendering classical probabilistic model checking algorithms inapplicable to reasoning about distributional specifications. This means that reasoning about distributional properties in MDPs requires new methods. The past decade has seen a rich line of theoretical work on analyzing Markov chains and MDPs as distribution transformers [31, 29, 11, 6, 4, 5, 2, 26]. However, existing automated methods are restricted to distributional reachability, safety, or reach-avoidance specifications.

In this paper, we present the first automated method for strategy verification and synthesis in MDPs with respect to *distributional ω-regular specifications*, significantly extending the class of distributional objectives for which automated methods are available. In doing so, we focus on the verification problem for a given MDP strategy, as well as the control problem which asks to synthesize an MDP strategy which ensures that a distributional ω-regular specification is satisfied. Our strategy verification and synthesis methods are based on the novel notion of *distributional certificates* which we introduce in this work. Distributional certificates provide a sound and complete proof rule for proving that an MDP under a distributionally memoryless strategy satisfies the distributional ω-regular specification of interest. We restrict our attention to *distributionally memoryless strategies*, which make moves based on the current distribution rather than the state, and which are known to be sufficient for reasoning about distributional reachability and safety [4, 5]. Our distributional certificates build on the ideas from program verification and certificates such as ranking function [22], invariants [21] or Büchi ranking functions [17], and bring these ideas to the setting of reasoning about distributional ω-regular specifications in MDPs.

We then present our automated algorithms for strategy verification and synthesis in MDPs with respect to distributional ω-regular specifications. In the setting of distributional objectives, it is known that safety and reachability are already hard, or more precisely, positivity and Skolem-hard [3]. The decidability of both are long-standing open problems in linear dynamical systems [32]. As a result, rather than aiming for sound and complete algorithms that would inherently be computationally expensive/infeasible, we adopt a template-based synthesis approach and instead design algorithms that can more efficiently search for distributional certificates and distributionally memoryless strategies that can be expressed in *affine arithmetic*. By fixing symbolic affine templates for the certificate and the strategy and by using existing methods for solving quantified formulas over real arithmetic, one is able to reduce the strategy verification and synthesis problems to satisfiability checking in existential theory of the reals, therefore obtaining sound algorithms that run in PSPACE. Furthermore, our algorithms also provide relative completeness guarantees for computing an affine distributional certificate and a memoryless strategy, whenever they exist.

We implement our approach and consider standard benchmarks and examples from the literature, while focusing on several distributional ω-regular specifications for our evaluation. Our results show the practicality of the approach and the potential for future applications.

Related work. In addition to the work already mentioned, we discuss a (non-exhaustive list of) few others. Our distributional certificates draw insights from classical certificates for program verification and template-based synthesis algorithms for their computation. Notable examples include ranking functions for proving termination in programs [22, 13] and invariants for proving safety in programs [21, 14]. Our distributional certificates draw insights from Büchi ranking functions of [17] for proving LTL properties in programs. However, there are several important differences. First, we leverage and lift the idea of Büchi ranking to the setting of probability distributions over MDP states. Second, our distributional certificates and algorithms for their computation also need to reason about *strategies* in MDPs. This is reflected in the following key difference. Our certificates provide soundness and completeness guarantees for all distributional ω-regular specifications and distributionally memoryless strategies, and proving this requires reasoning about reachability under a strategy (see the proof of Theorem 9). The certificate of [17], on the other hand, is complete only for specifications that can be represented via *deterministic* Büchi automata, if the specification needs to be satisfied from a set of initial states (see Corollary 2 in [17]).

In the distributional setting, the probabilistic logics defined in [10, 11, 2] are all orthogonal to the classical semantics, and the model checking techniques developed are not template-based. To the best of our knowledge, none of these works have been automated. The works [4, 5] propose certificates for distributional reachability, safety and reach-avoidance and design template-based synthesis algorithms for their computation. Our paper follows this line of work and introduces distributional certificates and template-based algorithms for *distributional ω-regular specifications*, hence significantly generalizing the class of distributional properties that we can automatically reason about.

Certificates were also used for reasoning about infinite-state probabilistic models such as probabilistic programs under the state-based view. In particular, supermartingale certificates were proposed for qualitative reachability [12, 15], quantitative reachability, safety and reach-avoidance [33, 19, 34, 18, 35], and most recently for qualitative ω-regular specifications [1]. However, these certificates are not, a priori, applicable to the distributional setting.

2 Preliminaries

In this section, we recall the basics of probabilistic systems and Markov decision processes. A *probability distribution* over a countable set X is a map $\mu : X \to [0, 1]$ such that $\sum_{x \in X} \mu(x) = 1$. The *support* of X is defined via $\text{supp}(\mu) = \{x \in X \mid \mu(x) > 0\}$. We use $\Delta(X)$ to denote the set of all probability distributions over X.

MDPs. A *Markov decision process (MDP)* is a tuple $\mathcal{M} = (S, Act, P)$. We use S to denote a finite set of *states* and Act to denote a finite set of *actions*. Slightly overloading the notation, for each state $s \in S$, we write $Act(s) \subseteq Act$ to denote the set of *available actions* at s. Finally, $P : S \times Act \to \Delta(S)$ is a *transition function*, assigning to each state s and available action $a \in Act(s)$ a probability distribution over the succcessor states. When $|Act(s)| = 1$ for each state s, we say that $\mathcal{M}$ is a *Markov chain*.

An *infinite path* in an MDP is a sequence $\rho = s_1, a_1, s_2, a_2, \cdots \in (S \times Act)^\omega$, such that $a_i \in Act(s_i)$ and $P(s_i, a_i)(s_{i+1}) > 0$ for all $i \in \mathbb{N}$. A *finite path* ϱ in an MDP is a finite prefix of an infinite path that ends in a state. We use ρ_i and ϱ_i to denote the i-th state along an (in)finite path. We use $\mathsf{IPaths}_\mathcal{M}$ and $\mathsf{FPaths}_\mathcal{M}$ to denote the sets of all infinite and finite paths in the MDP $\mathcal{M}$, respectively.

Semantics of MDPs. The semantics of MDPs are formalized in terms of strategies. A *strategy* (or *policy*) in an MDP $\mathcal{M}$ is a function $\pi : \mathsf{FPaths}_{\mathcal{M}} \to \Delta(Act)$ which to each finite path (called a *history*) assigns a probability distribution over the action to be taken next. We require that, if a finite path $\varrho \in \mathsf{FPaths}_{\mathcal{M}}$ ends in a state s, then $\mathrm{supp}(\pi(\varrho)) \subseteq Act(s)$. A strategy is said to be *memoryless* if the probability distribution over actions depends only on the last state of the finite path and not on the whole history, i.e. if $\pi(\varrho) = \pi(\varrho')$ whenever ϱ and ϱ' end in the same state. For every initial state distribution $\mu_0 \in \Delta(S)$, an MDP $\mathcal{M}$ and a strategy π together give rise to a probability space over the set of all infinite paths in the MDP [8]. We denote by $\mathbb{P}^{\pi}_{\mu_0}$ the probability measure and by $\mathbb{E}^{\pi}_{\mu_0}$ the expectation operators in this probability space, omitting the MDP $\mathcal{M}$ from the notation when clear from the context.

MDPs as distribution transformers. MDPs are typically regarded as random generators of infinite paths, giving rise to a probability space over the set of all infinite paths in the MDP. Classical probabilistic model checking problems then explore the expected behaviour of these randomly generated infinite paths, giving rise to *path properties* [8]. However, one can also view MDPs as *(deterministic) transformers of distributions.*

Consider an MDP $\mathcal{M}$, a strategy π, and an initial state distribution $\mu_0 \in \Delta(S)$. For each $i \in \mathbb{N}$ and state s, define $\mu_i(s) = \mathbb{P}^{\pi}_{\mu_0}[\rho \in \mathsf{IPaths}_{\mathcal{M}} \mid \rho_i = s]$, i.e. the probability that the i-th state of a randomly generated infinite path is s. We write $\mu_i = \mathcal{M}^{\pi}(\mu_0, i)$ for the induced probability distribution of the i-th state of a randomly generated infinite path. Hence, the MDP $\mathcal{M}$, a strategy π, and an initial state distribution $\mu_0 \in \Delta(S)$ together give rise to a sequence of probability distributions over the MDP states

$$\mu_0, \quad \mu_1 = \mathcal{M}^{\pi}(\mu_0, 1), \quad \mu_2 = \mathcal{M}^{\pi}(\mu_0, 2), \quad \mu_3 = \mathcal{M}^{\pi}(\mu_0, 3), \quad \dots$$

One can then study properties of this sequence of distributions. Some examples are *distributional reachability* and *distributional safety*, which ask if the induced sequence of distributions contains or does not contain an element of some specified set of distributions [4, 5].

ω-regular specifications. In this work we will consider ω-regular specifications, which subsume a broad class of specifications such as those belonging to linear temporal logic (LTL) or computation tree logic (CTL) [8]. An *ω-regular specification* over a finite set AP of atomic propositions is defined by a *non-deterministic Büchi automaton (NBA)* $N = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, $\Sigma = 2^{\mathsf{AP}}$ is a finite set of letters, $\delta : Q \times \Sigma \to 2^Q$ is a (non-deterministic) transition function, $q_0 \in Q$ is the initial state and $F \subseteq Q$ are accepting states. An infinite word of letters $\sigma_1, \sigma_2, \dots \in \Sigma^{\omega}$ is said to be *accepting*, if it gives rise to at least one accepting run in N, i.e. if there exists a run $q_0, q_1, q_2, \dots$ such that $q_{i+1} \in \delta(q_i, \sigma_i)$ for each i and such that $q_i \in F$ for infinitely many i. Note that given an LTL formula φ, it can be converted to an equivalent NBA N^{φ} in exponential time (see e.g., [8]). In what follows, we will often write examples and benchmarks in LTL as it will be easier and often more intuitive. But for our analysis, we will convert them to their equivalent NBA and reason only about these NBA as the ω-regular specification.

Transition systems. In order to reason about the synchronous evolution of a sequence of distributions over the MDP states and a run in the NBA, we will later introduce the notion of product distributional transition systems in Section 4. Hence, we here recall the notion of transition systems, which are commonly used to model imperative numerical programs.

An *(infinite-state) transition system* is a tuple $\mathcal{T} = (L, V, l_{\mathrm{init}}, \theta_{\mathrm{init}}, \mapsto)$, where

- L is a finite set of locations,
- V is a finite set of real-valued variables,

- $l_{\text{init}} \in L$ is the initial location,
- $\theta_{\text{init}} \subseteq \mathbb{R}^{|V|}$ is the set of initial variable valuations, and
- $\mapsto$ is a finite set of transitions of the form $\tau = (l_\tau, l'_\tau, G_\tau, U_\tau)$ with l_τ a source location, l'_τ a target location, G_τ a guard which is a boolean predicate over the variables in V, and $U_\tau : \mathbb{R}^n \to \mathbb{R}^n$ an update function.

A *state* in the transition system is a tuple $(l, x) \in L \times \mathbb{R}^{|V|}$ consisting of a location in L and a valuation of variables in V. A transition $\tau = (l_\tau, l'_\tau, G_\tau, U_\tau)$ is said to be *enabled* at a state (l, x) if $l = l_\tau$ and $x \models G_\tau$. An *infinite path* (or a *run)* in the transition system is a sequence of states $(l_0, x_0), (l_1, x_1), \dots$ with $l_0 = l_{\text{init}}$, $x_0 \in \mathsf{Init}$, and where for each $i \in \mathbb{N}_0$ there exists a transition $\tau_i = (l_i, l_{i+1}, G_\tau, U_\tau)$ enabled at (l_i, x_i) such that $x_{i+1} = U_\tau(x_i)$. A state (l, x) is said to be *reachable*, if there exists an infinite path that contains (l, x).

3 Problem Statement

We now formally define the problems that we consider in this work. Our goal is to design fully automated algorithms for formal verification and control in MDPs with respect to distributional ω-regular specifications. Hence, we first need to formalize the notion of distributional ω-regular specifications. In what follows, let $\mathcal{M} = (S, Act, P)$ be an MDP.

Distributional ω-regular specifications. Similarly to the classical ω-regular specification setting, we first need to specify a finite set of *atomic propositions* AP. We are interested in reasoning about a sequence of distributions induced by an MDP under a strategy. Hence, we let the set AP consist of finitely many logical formulas of the form $\mathsf{exp}(\mu(s_1), \dots, \mu(s_{|S|})) \geq 0$. Here, $\mathsf{exp} : \mathbb{R}^{|S|} \to \mathbb{R}$ is an arithmetic expression over the probabilities $\mu(s_1), \dots, \mu(s_{|S|})$ of being in each state of the MDP, where $s_1, \dots, s_{|S|}$ is an arbitrary (but throughout fixed) enumeration of MDP states. In practice, we let AP contain exactly those atomic propositions that appear in the property that we want to reason about. A *distributional ω-regular specification* φ is then defined by an NBA $N^\varphi = (Q, \Sigma, \delta, q_0, F)$ with $\Sigma = 2^{\mathsf{AP}}$.

We now define the semantics of distributional ω-regular specifications. Consider a finite set of atomic propositions AP, a distributional ω-regular specification φ, a strategy π and an initial distribution $\mu_0 \in \Delta(S)$ in the MDP $\mathcal{M}$. The MDP $\mathcal{M}$ under strategy π from the initial distribution μ_0 induces an infinite word $\sigma_0, \sigma_1, \sigma_2, \dots$ in the language 2^{AP} as follows. As defined in Section 2, an MDP $\mathcal{M}$ under strategy π from the initial distribution μ_0 induces a sequence $\mu_0, \mu_1, \mu_2, \dots$ of distributions over MDP states. Then, for each $i \in \mathbb{N}_0$, we define the letter σ_i as the set of all atomic propositions in AP that are satisfied at the distribution μ_i, i.e. $\sigma_i = \{p \in \mathsf{AP} \mid \mu_i \models p\}$, where we use $\models$ to denote proposition satisfaction. We say that the MDP $\mathcal{M}$ *satisfies* distributional ω-regular specification φ under strategy π from initial distribution $\mu_0 \in \Delta(S)$, if this infinite word $\sigma_0, \sigma_1, \sigma_2, \dots$ is accepted by the NBA N^φ.

Distributionally memoryless strategies. We restrict our attention to a class of strategies called distributionally memoryless strategies. A strategy $\pi : \mathsf{FPaths}_\mathcal{M} \to \Delta(Act)$ is said to be *distributionally memoryless* if the probability distribution over actions prescribed by the strategy depends only on the current distribution over the MDP states and not on the whole history. Formally, we require that for any initial distribution $\mu_0 \in \mathsf{Init}$ and for any two finite runs $\rho = s_0, a_0, s_1, a_1, \dots, s_n$ and $\rho' = s'_0, a'_0, s'_1, a'_1, \dots, s'_n$ that induce the sequences of probability distributions $\mu_0, \mu_1, \dots, \mu_n$ and $\mu'_0, \mu'_1, \dots, \mu'_n$ with $\mu_n = \mu'_n$, we have $\pi(\rho) = \pi(\rho')$. When the strategy π is distributionally memoryless, we write $\mathcal{M}^\pi(\mu) = \mathcal{M}^\pi(\mu, 1)$ to denote an application of a single-step distribution transformer operator.

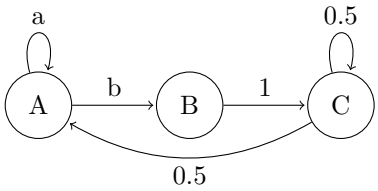


Figure 1 An MDP which will serve as our running example. The MDP contains three states $S = \{A, B, C\}$, two actions $Act = \{a, b\}$ with $Act(A) = \{a, b\}$, $Act(B) = \{a\}$, $Act(C) = \{a\}$, and its transition function is defined via $P(A, a)(A) = 1$, $P(A, b)(B) = 1$, $P(B, a)(C) = 1$, $P(C, a)(C) = P(C, a)(A) = 0.5$. We consider a singleton initial distribution set $\mathsf{Init} = \{(A : \frac{1}{3},\ B : \frac{1}{3},\ C : \frac{1}{3})\}$.

It was shown in [4, 5] that distributionally memoryless strategies are sufficient for reasoning about distributional reachability, safety and reach-avoid specifications. That is, for each of these distributional specifications, there exists a strategy in the MDP under which the specification is satisfied if and only if there exists a distributionally memoryless strategy in the MDP under which the specification is satisfied. While this result need not necessarily hold for distributional ω-regular specifications, the restriction will be needed for enabling automated verification and synthesis as they can be represented in a more compact form.

Problem statement. We are now ready to define our strategy verification and synthesis problems. Consider an MDP $\mathcal{M}$, a set of initial distributions $\mathsf{Init} \subseteq \Delta(S)$, and a distributional ω-regular specification φ:

1. **Strategy verification problem.** Given a distributionally memoryless strategy π, verify that the MDP $\mathcal{M}$ satisfies φ under π from every initial distribution $\mu_0 \in \mathsf{Init}$.
2. **Strategy synthesis problem.** Compute a distributionally memoryless strategy π, such that the MDP $\mathcal{M}$ satisfies φ under π from every initial distribution $\mu_0 \in \mathsf{Init}$.

▶ **Example 1** (Running example)**.** The MDP shown in Fig. 1 was considered in [4] for studying distributional safety specifications and it will serve as our running example. We consider the strategy synthesis and verification problems with respect to the distributional ω-regular specification $\varphi = \mathbf{G}\,\mathbf{F}\,(p(B) \geq 0.249)$. For readability, we specify φ as an LTL formula over the set of atomic propositions $\mathsf{AP} = \{(\mu(B) \geq 0.249)\}$. This is an example of a *distributional persistence specification*, which specifies that the sequence of distributions $\mu_0, \mu_1, \ldots$ should contain infinitely many distributions μ_i with $\mu_i(B) \geq 0.249$. The goal of strategy synthesis is to compute a strategy under which φ is satisfied. An example of such a strategy is a (distributionally) memoryless strategy π which in state A takes action b with probability 1. The goal of strategy verification is to verify this claim.

▶ Remark 2 (Problem hardness). The problem of determining if an MDP $\mathcal{M}$ under a distributionally memoryless strategy π satisfies a distributional ω-regular specification φ is computationally hard. It was shown to be Skolem-hard already in the very restricted setting when $\mathcal{M}$ is a Markov chain (so the strategy π is trivial) and φ is a distributional reachability specification for an affine set of goal distributions $\{\mu \in \Delta(S) \mid \mu(s_1) = 0.25\}$ [3].

▶ Remark 3 (Universal and existential satisfaction problems). In the terminology of [5] which considered distributional reachability and safety specifications, our problem corresponds to the *universal satisfaction* setting, where the specification needs to be satisfied from *every* initial distribution $\mu_0 \in \mathsf{Init}$. Dually, one can also consider the *existential satisfaction* setting, where the specification needs to be satisfied from *at least one* initial distribution $\mu_0 \in \mathsf{Init}$. While we will focus on the universal satisfaction setting for ease of presentation, we also show that all our results straightforwardly extend to the existential satisfaction setting as well.

▶ Remark 4 (Memoryless vs distributionally memoryless strategies). Note that distributionally memoryless strategies are *not necessarily* memoryless (in the "classical" state-based sense). This fact was already shown in [4] for distributional safety specifications, where one may require infinite memory as well as randomized strategies in order to satisfy the specification. This is in stark contrast with the state-based view, where deterministic memoryless strategies are sufficient for specifications such as reachability and safety [8].

4 Certificate for Distributional ω-regular Specifications

We now present our sound and complete certificate for proving that an MDP under a distributionally memoryless strategy satisfies some distributional ω-regular specification, which is the main theoretical contribution of this work. In Section 5, we will present our algorithms for automated synthesis and verification of strategies in MDPs with respect to distributional ω-regular specifications, where the certificate will play a central role.

In what follows, we fix an MDP $\mathcal{M} = (S, Act, P)$, a set of initial distributions Init, a distributionally memoryless strategy π, and a distributional ω-regular specification φ defined over atomic propositions AP with an NBA $N^\varphi = (Q, \Sigma, \delta, q_0, F)$ where $\Sigma = 2^{\mathsf{AP}}$.

4.1 Product Distributional Transition System

Recall from Section 2 that, for each initial distribution in Init, the MDP $\mathcal{M}$ and the strategy π induce a sequence of distributions over the MDP states. This sequence gives rise to an infinite word in the language 2^{AP} and a run in the NBA N^φ. In what follows, we introduce product distributional transition systems (PDTS), which will allow us to synchronously reason about the distribution sequence and the NBA run.

▶ **Definition 5** (Product distributional transition system). *Let $\mathcal{M} = (S, Act, P)$ be an MDP, Init be a set of initial distributions, π be a distributionally memoryless strategy, and $N^\varphi = (Q, 2^{\mathsf{AP}}, \delta, q_0, F)$ be an NBA for some distributional ω-regular specification φ defined over atomic propositions AP. A product distributional transition system (PDTS) is a transition system $\mathcal{T}^\times = (L^\times, V^\times, l^\times_{init}, \theta^\times_{init}, \mapsto^\times)$, where*

- *$L^\times = Q$ is the set of states of N^φ;*
- *$V^\times = \{\mu_1, \ldots, \mu_{|S|}\}$ is a finite state of real-valued variables, with each variable μ_i corresponding to the probability of being in an MDP state s_i;*
- *$l^\times_{init} = q_0$ is the initial state of N^φ;*
- *$\theta^\times_{init} = \mathsf{Init}$ is the set of initial distributions in $\mathcal{M}$; and*
- *$\mapsto^\times = \{(q, q', G(\sigma), \mathcal{M}^\pi) \mid q, q' \in Q, \sigma \in 2^{\mathsf{AP}}, q' \in \delta(q, \sigma)\}$, where $G(\sigma) = (\wedge_{p \in \sigma} p) \wedge (\wedge_{p \in \mathsf{AP} \setminus \sigma} \neg p)$ is the predicate defined by atomic propositions contained in σ, and $\mathcal{M}^\pi$ is the linear function defined by the single-step distribution transformer operator of $\mathcal{M}$ and π.*

▶ **Example 6.** Fig. 2 left shows the NBA for the distributional specification $\varphi = \mathbf{G}\,\mathbf{F}\,(p(B) \geq 0.249)$ considered in Example 1. Fig. 2 right then shows the PDTS of our running example MDP in Fig. 1 and the NBA. The PDTS has the same set of locations $L^\times = \{q_0, q_1\}$ as the NBA and the set of variables $V^\times = \{\mu_1, \mu_2, \mu_3\}$ corresponding to the probabilities of being in MDP states A, B, C. The initial location is $l_{\text{init}} = q_0$ and the set of initial distributions is $\mathsf{Init} = \{(A : \frac{1}{3},\ B : \frac{1}{3},\ C : \frac{1}{3})\}$. Finally, the three PDTS transitions are shown in Fig. 2.

Note that PDTS indeed models a synchronous execution of a sequence of distributions over MDP states and a run in the NBA. Each infinite path $(q_0, \mu_0), (q_1, \mu_1), \ldots$ in the PDTS starts from a state $(q_0, \mu_0) \in \{q_0\} \times \mathsf{Init}$. Then, for each state (q_i, μ_i) along the infinite path,

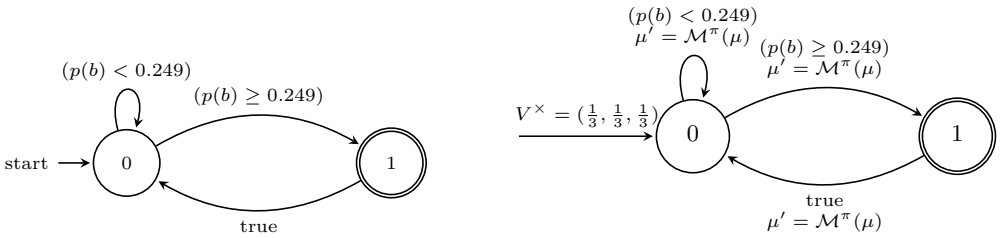


Figure 2 The figure on the left shows the NBA for distributional specification $\varphi = \mathrm{GF}(p(b) \geq 0.249)$. The figure on the right then shows the PDTS of the MDP in Fig. 1 with the strategy that in state A takes action b with probability 1, and the NBA on the left. Each PDTS transition is labeled with its guard (top line) and its update function (bottom line). We write $\mu' = \mathcal{M}^\pi(\mu)$ as a shorthand notation for $\mu' = \{A : 0.5 \cdot \mu(C), B : \mu(A), C : \mu(B) + 0.5 \cdot \mu(C)\}$.

the next state is obtained by applying some enabled transition $(q_i, q_{i+1}, G(\sigma_i), \mathcal{M}^\pi)$. For the transition to be enabled, we must have $\sigma_i = \{p \in \mathsf{AP} \mid \mu_i \models p\}$ be the unique letter defined by all atomic propositions satisfied in μ_i. The PDTS then moves to a state (q_{i+1}, μ_{i+1}), where $\mu_{i+1} = \mathcal{M}^\pi(\mu_i)$ is indeed the next distribution in the sequence and $q_{i+1} \in \delta(q_i, \sigma_i)$ is indeed a successor state in the NBA.

An infinite path $(q_0, \mu_0), (q_1, \mu_1), \ldots$ in the PDTS is *accepting* if $(q_i, \mu_i) \in F \times \Delta(S)$ for infinitely many $i \in \mathbb{N}_0$, i.e. if it visits states with locations belonging to the accepting set of the NBA infinitely often. The following proposition formalizes the relationship between the satisfaction of a distributional ω-regular specification in the MDP and the existence of an accepting infinite path in the PDTS. We state the proposition for a single initial distribution $\mu_0 \in \mathsf{Init}$, so that it is applicable both in the universal and the existential satisfaction problem settings (see Remark 3).

▶ **Proposition 7.** *An MDP $\mathcal{M}$ with an initial distribution $\mu_0 \in \Delta(S)$ satisfies a distributional ω-regular specification φ under a distributionally memoryless strategy π if and only if there exists an accepting infinite path in the PDTS $\mathcal{T}^\times = (L^\times, V^\times, l^\times_{init}, \theta^\times_{init}, \mapsto^\times)$.*

Proof. Suppose that MDP $\mathcal{M}$ with initial distribution μ_0 satisfies distributional ω-regular specification φ under distributionally memoryless strategy π. Let $\mu_0, \mu_1, \ldots$ be the sequence of distributions induced by the MDP $\mathcal{M}$ under strategy π, and let $\sigma_0, \sigma_1, \ldots$ be the induced infinite word from the initial distribution μ_0. By the definition of satisfiability of distributional ω-regular specifications in Section 3, the infinite word $\sigma_0, \sigma_1, \ldots$ is accepted by the NBA N^φ. Hence, there exists a run $q_0, q_1, \ldots$ in N^φ such that $q_{i+1} \in \delta(q_i, \sigma_i)$ for each $i \in \mathbb{N}$. But this also means that $(q_0, \mu_0), (q_1, \mu_1), \ldots$ is an accepting path in the PDTS $\mathcal{T}^\times = (L^\times, V^\times, l^\times_{\text{init}}, \theta^\times_{\text{init}}, \mapsto^\times)$, which proves one direction of the proposition.

Conversely, suppose that there exists an accepting infinite path $(q_0, \mu_0), (q_1, \mu_1), \ldots$ in the PDTS $\mathcal{T}^\times = (L^\times, V^\times, l^\times_{\text{init}}, \theta^\times_{\text{init}}, \mapsto^\times)$. Then, by the definition of transition update functions in the PDTS, we know that $\mu_0, \mu_1, \ldots$ is the sequence of distributions induced by the MDP $\mathcal{M}$ under strategy π. Moreover, by the definition of transition guards in the PDTS, we know that $q_{i+1} \in \delta(q_i, \sigma_i)$ for each $i \in \mathbb{N}$ with σ_i being the unique letter defined by atomic propositions in AP satisfied in μ_i. Hence, $q_0, q_1, \ldots$ is an infinite run in the NBA N^φ induced by the infinite word $\sigma_0, \sigma_1, \ldots$. But from the fact that $(q_0, \mu_0), (q_1, \mu_1), \ldots$ is an accepting run in the PDTS, it follows that $q_i \in F$ for infinitely many $i \in \mathbb{N}$ and so the infinite word $\sigma_0, \sigma_1, \ldots$ is accepted by the NBA N^φ. By the definition of satisfiability of distributional ω-regular specifications in Section 3, this implies that MDP $\mathcal{M}$ with initial distribution μ_0 satisfies specification φ under strategy π, which concludes our proof. ◀

4.2 Distributional Certificates

We are now ready to define our notion of distributional certificates. A *distributional certificate* is a pair $(\mathcal{C}, I)$ that consists of two components – a *distributional Büchi ranking function* $\mathcal{C}$ and a *distributional invariant* I. The distributional Büchi ranking function $\mathcal{C} : Q \times \Delta(S) \to \mathbb{R}$ is a function that to each state of the PDTS assigns a real value, which is required to satisfy two conditions. First, the *Initial condition* requires the value of $\mathcal{C}$ to be non-negative at all initial states of the PDTS. Second, the *Büchi ranking condition* requires that, for every reachable state in the PDTS at which the value of $\mathcal{C}$ is non-negative, there exists at least one successor state at which non-negativity is preserved. Furthermore, the value of $\mathcal{C}$ decreases by at least 1 if the state is not contained in the accepting set of the PDTS. We prove in Theorem 9 below that these two conditions are necessary and sufficient to ensure that, for every initial state (q_0, μ_0) in the PDTS, there exists an accepting infinite path in the PDTS.

Note that the Büchi ranking condition needs to be satisfied only at reachable states of the PDTS. However, the problem of determining the exact set of reachable states is not feasible. Hence, with later automation in mind, we append our certificate with a distributional invariant $I \subseteq Q \times \Delta(S)$, which is a set that over-approximates the set of reachables states in the PDTS. This is ensured by extending the Initial condition to require that all initial states of the PDTS are contained in the invariant I, and by extending the Büchi ranking condition to require that the successor state described above is also contained in the invariant I.

The following definition formalizes this intuition. In what follows, for each letter $\sigma \in 2^{\mathsf{AP}}$ and distribution $\mu \in \Delta(S)$, we write $\mu \models G(\sigma)$ as a shorthand for $\mu \models (\wedge_{p \in \sigma} p) \wedge (\wedge_{p \in \mathsf{AP} \setminus \sigma} \neg p)$.

▶ **Definition 8** (Distributional certificate). *A distributional certificate for an MDP $\mathcal{M}$ with a set of initial distributions* Init*, a distributionally memoryless strategy π, and a distributional ω-regular specification φ with NBA N^{φ}, is a tuple $(\mathcal{C}, I)$ consisting of a function $\mathcal{C} : Q \times \Delta(S) \to \mathbb{R}$ and a set $I \subseteq Q \times \Delta(S)$, such that the following conditions hold:*

- Initial condition. *For all $\mu_0 \in$ Init, we have $\mathcal{C}(q_0, \mu_0) \geq 0$ and $(q_0, \mu_0) \in I$.*
- Büchi ranking condition. *We have the following:*
  - Non-negativity at accepting states. *For all NBA states $q \in F$ and letters $\sigma \in 2^{\mathsf{AP}}$,*

$$\forall \mu \in \mathbb{R}^{|S|}. \bigvee_{q' \in \delta(q,\sigma)} \mu \in \Delta(S) \wedge \mu \models G(\sigma) \wedge \mathcal{C}(q, \mu) \geq 0 \wedge (q, \mu) \in I \implies \mathcal{C}(q', \mathcal{M}^{\pi}(\mu)) \geq 0 \wedge (q', \mathcal{M}^{\pi}(\mu)) \in I. \tag{1}$$

  - Strict decrease and non-negativity at non-accepting states. *For all NBA states $q \notin F$ and letters $\sigma \in 2^{\mathsf{AP}}$,*

$$\forall \mu \in \mathbb{R}^{|S|}. \bigvee_{q' \in \delta(q,\sigma)} \mu \in \Delta(S) \wedge \mu \models G(\sigma) \wedge \mathcal{C}(q, \mu) \geq 0 \wedge (q, \mu) \in I \implies \mathcal{C}(q, \mu) - 1 \geq \mathcal{C}(q', \mathcal{M}^{\pi}(\mu)) \geq 0 \wedge (q', \mathcal{M}^{\pi}(\mu)) \in I. \tag{2}$$

The following theorem establishes that distributional certificates provide a sound and complete proof rule for proving that an MDP under a distributionally memoryless strategy satisfies a distributional ω-regular specification.

▶ **Theorem 9** (Soundness and completeness). *An MDP $\mathcal{M}$ with a set of initial distributions* Init *under a distributionally memoryless strategy π satisfies a distributional ω-regular specification φ if and only if there exists a distributional certificate for $\mathcal{M}$,* Init*, π and φ.*

Proof.

Soundness. Suppose that there exists a distributional certificate $(\mathcal{C}, I)$ for $\mathcal{M}$, Init, π and φ. To show that φ is satisfied, by Proposition 7 it suffices to show that the PDTS $\mathcal{T}^{\times}$ admits an accepting infinite path for every initial state in $\{q_0\} \times \mathsf{Init}$.

Fix an initial state $(q_0, \mu_0) \in \{q_0\} \times \mathsf{Init}$. By the Initial condition in Definition 8, we know that $\mathcal{C}(q_0, \mu_0) \geq 0$ and $(q_0, \mu_0) \in I$. Hence, by the Büchi ranking condition in Definition 8, we can repeadly select successor states in order to obtain an infinite path $(q_0, \mu_0), (q_1, \mu_1), \ldots$ in $\mathcal{T}^{\times}$ such that, for each $i \in \mathbb{N}_0$, we have

- $\mathcal{C}(q_i, \mu_i) \geq 0$ and $(q_i, \mu_i) \in I$, and
- whenever $q_i \notin F$ is not an accepting state in N^{φ}, we have $\mathcal{C}(q_i, \mu_i) - 1 \geq \mathcal{C}(q_{i+1}, \mu_{i+1})$.

We claim that $(q_0, \mu_0), (q_1, \mu_1), \ldots$ is an accepting infinite path in $\mathcal{T}^{\times}$. To prove this, note that for every (q_i, μ_i) with $q_i \notin F$, the value of $\mathcal{C}$ needs to keep decreasing by at least 1 in each subsequent step while also remaining non-negative. Hence, in at most $\lceil \mathcal{C}(q_i, \mu_i) \rceil$ steps, the path must again reach an accepting state. Thus, the infinite path $(q_0, \mu_0), (q_1, \mu_1), \ldots$ reaches accepting states in $F \times \Delta(S)$ infinitely many times and is an accepting infinite path. Since the initial state $(q_0, \mu_0) \in \{q_0\} \times \mathsf{Init}$ was arbitrary, this concludes the proof.

Completeness. Conversely, suppose that $\mathcal{M}$ with a set of initial distributions Init under distributionally memoryless strategy π satisfies distributional ω-regular specification φ. We construct an instance $(\mathcal{C}, I)$ of a distributional certificate for $\mathcal{M}$, Init, π and φ as follows.

Consider an arbitrary but throughout fixed enumeration $q_1, \ldots, q_{|Q|}$ of NBA states. We define an operator $\textsc{Next} : Q \times \Delta(S) \to Q \times \Delta(S)$ via

- $\textsc{Next}(q, \mu) = (q_i, \mathcal{M}^{\pi}(\mu))$ with i being the smallest index such that $(q, \mu), (q_i, \mathcal{M}^{\pi}(\mu))$ are successor states along some accepting infinite path in the PDTS, if such an accepting infinite path exists, or
- $\textsc{Next}(q, \mu) = (q, \mu)$, otherwise.

In other words, $\textsc{Next}(q, \mu)$ fixes a successor state of (q, μ) along some accepting infinite path in the PDTS if such a path exists, or halts the sequence at the state (q, μ) otherwise. Therefore, the transitive closure of the operator $\textsc{Next}(q, \mu)$ from the set of initial PDTS states $\{q_0\} \times \mathsf{Init}$ allows us to consistently fix a *unique accepting infinite path* for each state (q, μ) that is contained along some accepting infinite path.

We can now define our distributional certificate $(\mathcal{C}, I)$. Let distributional invariant $I \subseteq Q \times \Delta(S)$ be the set of all states in the PDTS that are reachable from $\{q_0\} \times \mathsf{Init}$ under the transitive closure of the $\textsc{Next}$ operator. Moreover, for each PDTS state $(q, \mu) \in I$, let $d_{\text{accept}}(q, \mu)$ denote the number of steps when applying the $\textsc{Next}$ operator until an accepting state in $F \times \Delta(S)$ is reached, with $d_{\text{accept}}(q, \mu) = 0$ if $(q, \mu) \in F \times \Delta(S)$ is an accepting state. Finally, let the distributional Büchi ranking function $\mathcal{C}$ be defined via

$$\mathcal{C}(q, \mu) = \begin{cases} d_{\text{accept}}(q, \mu), & \text{if } (q, \mu) \in I, \\ -1, & \text{otherwise.} \end{cases}$$

We claim that $(\mathcal{C}, I)$ is an instance of a distributional certificate. The Initial condition in Definition 8 is satisfied since the MDP $\mathcal{M}$ under strategy μ satisfies φ, therefore every initial state $(q_0, \mu) \in \{q_0\} \times \mathsf{Init}$ belongs to some accepting infinite path in the PDTS and so $(q_0, \mu) \in I$ and $d_{\text{accept}}(q_0, \mu) \geq 0$. On the other hand, by the definition of the $\textsc{Next}$ operator and I, for each $(q, \mu) \in I$ we have that also $\textsc{Next}(q, \mu) \in I$. Moreover, $\textsc{Next}(q, \mu) = d_{\text{accept}}(q, \mu) - 1 \geq 0$ if $q \notin F$ and $\textsc{Next}(q, \mu) \geq 0$ id $q \in F$. Hence, the Büchi ranking condition in Definition 8 is also satisfied, and $(\mathcal{C}, I)$ is a distributional certificate. ◀

▶ **Example 10.** Consider again the MDP in Fig. 1. As in Example 1, consider the strategy π which in state A takes action b with probability 1, and the distributional specification $\varphi = \mathbf{G}\,\mathbf{F}\,(p(B) \geq 0.249)$. An NBA for φ and the resulting PDTS are shown in Fig. 2. The following is an example of a distributional certificate $(\mathcal{C}, I)$ for $\mathcal{M}$, Init, π and φ:

$$\mathcal{C}(q,\mu) = \begin{cases} 1 + 250 \cdot \mu(A) + 750 \cdot \mu(C), & \text{if } q = q_0, \\ 1.25 - 1.25 \cdot \mu(C), & \text{if } q = q_1, \end{cases}$$

and $I = \{(q_0, \mu) \mid 1.25 + \mu(A) + 1 \cdot \mu(B) - \mu(C) \geq 0\} \cup \{(q_1, \mu) \mid \mu(A) + 0.25 \cdot \mu(B)\}$.

One can verify by inspection that the Initial condition and the Büchi ranking condition in Definition 8 are satisfied. We note that the above distributional certificate $(\mathcal{C}, I)$ is the certificate computed by our prototype implementation in Section 6.

▶ Remark 11 (Distributional certificates for the existential problem). As discussed in Section 2 and Remark 3, our distributional certificate in Definition 4.2 and our soundness and completeness result in Theorem 9 consider the universal satisfaction setting. However, their extension to the existential setting is immediate. The only required change in the definition of distributional certificates is to require the Initial condition in Definition 4.2 to hold *for some* initial distribution $\mu_0 \in \mathsf{Init}$. On the other hand, the soundness and completeness proof proceeds analogously as in the proof of Theorem 9, with the difference in the completeness proof being that the distributional invariant I is defined as the transitive closure of the NEXT operator with the singleton initial set $\{(q_0, \mu_0)\}$, rather than the initial set $\{q_0\} \times \mathsf{Init}$.

5 Template-based Strategy Verification and Synthesis with Certificates

We now present our algorithms for the strategy verification and synthesis problems for distributional ω-regular specifications. The core of the verification algorithm is to synthesize an *affine* distributional certificate in the PDTS of the input MDP and the specification, which proves that the specification is satisfied. When we move from strategy verification to strategy synthesis, we also synthesize an *affine* distributionally memoryless strategy.

The restriction to affine distributional certificates and affine distributionally memoryless strategies is needed to ensure tractability. While Theorem 9 establishes soundness and completeness of our distributional certificates, in combination with Remark 2 it also implies that giving a sound and complete algorithm for synthesizing distributional certificates is Skolem-hard. Hence, in this section, we instead focus on designing sound and *relatively complete* algorithms for synthesizing an affine distributional certificate together with an affine distributionally memoryless strategy (the latter for the synthesis problem), when they exist.

Affine distributional certificates and strategies. We first formalize the notions of affine distributional certificates and distributionally memoryless strategies:

- **Affine distributional certificates.** A distributional certificate $(\mathcal{C}, I)$ is said to be *affine* if both the distributional Büchi ranking function $\mathcal{C}$ and the distributional invariant I can be expressed in terms of affine expressions and inequalities over the space $\Delta(S)$ of probability distributions over MDP states. We require $\mathcal{C}$ to be of the form

$$\mathcal{C}(q,\mu) = \sum_{i=1}^{|S|} a_i^q \cdot \mu(s_i) + b^q, \tag{3}$$

where $a_1^q, \ldots, a_{|S|}^q, b^q$ are some real valued coefficients for each NBA state $q \in Q$. That is, for each NBA state q, the function $\mathcal{C}(q, \cdot)$ is an affine function over the probabilities

of being in each MDP state, with $\mu(s_1), \ldots, \mu(s_{|V|})$ being the variables that capture probabilities of being in each MDP state and $a^q_1, \ldots, a^q_{|S|}, b^q$ being the coefficients of the affine function. Similarly, we require I to be a set defined by a conjunction of N_I affine inequalities over the probabilities of being in each MDP state, i.e. to be of the form

$$I = \Big\{(\mu, q) \in \Delta(S) \times Q \mid \wedge_{k=1}^{N_I} I^k(q, \mu) \geq 0\Big\}, \tag{4}$$

where each $I^k(q, \mu) = \sum_{i=1}^{|S|} c_i^{k,q} \cdot \mu(s_i) + d^{k,q} \geq 0$ and $c_1^{k,q}, \ldots, c_{|S|}^{k,q}, d^{k,q}$ are some real valued coefficients for each NBA state $q \in Q$ and each $k \in \{1, \ldots, N_I\}$. The number N_I is referred to as the *size of the invariant* and will be an algorithm parameter.

- **Affine distributionally memoryless strategies.** A distributionally memoryless strategy $\pi : \Delta(S) \to \Delta(Act)$ is said to be *affine*, if for each state $s \in S$, action $a \in Act$ and state distribution $\mu \in \Delta(S)$, the probability of taking action a in state s given the current distribution over states μ is of the form

$$\pi(s, a)(\mu) = \frac{\sum_{i=1}^{|S|} e^{\pi}_{i,s,a} \cdot \mu(s_i) + f^{\pi}_{s,a}}{\sum_{i=1}^{|S|} g^{\pi}_{i,s} \cdot \mu(s_i) + h^{\pi}_s}, \tag{5}$$

where $e^{\pi}_{1,s,a}, \ldots, e^{\pi}_{|S|,s,a}, f^{\pi}_{s,a}$ and $g^{\pi}_{1,s}, \ldots, g^{\pi}_{|S|,s}, h^{\pi}_s$ are real valued constants. The denominator is used in order to normalize the probabilities such that the sum of probabilities of all actions being taken at a state s is 1.

Algorithm input. Both our verification and synthesis algorithms take as input an MDP $\mathcal{M} = (S, Act, P)$, a set of initial distributions Init, and a distributional ω-regular specification φ defined over atomic propositions AP. We assume that the distributional ω-regular specification is provided via an NBA $N^{\varphi} = (Q, \Sigma, \delta, q_0, F)$ with letters $\Sigma = 2^{\mathsf{AP}}$, which accepts the same set of infinite words over 2^{AP} as φ. Finally, the algorithms also take as input the size of the invariant N_I that needs to be synthesized. The verification algorithm in addition takes as input an affine distributionally memoryless strategy π.

Algorithm overview. Both verification and synthesis algorithms follow a template-based synthesis approach and proceed in four steps. In the first step, the PDTS of the input MDP and the distributional specification is constructed. In the second step, the algorithms fix a symbolic template for the affine distributional certificate, i.e. symbolic variables for each real valued coefficient in affine expressions that define the certificate. The synthesis algorithm also fixes a symbolic template for the affine distributionally memoryless strategy. In the third step, the algorithms collect a system of constraints over the symbolic template variables, that together encode all defining conditions of distributional certificates in Definition 8 as well as conditions for the strategy template to define a valid distributionally memoryless strategy (the latter for the synthesis algorithm). Finally, in the fourth step, the collected system of constraints is solved by using an SMT-solver, to get a concrete valuation of the symbolic template variables which in turn gives rise to a distributional certificate and a distributionally memoryless strategy. In what follows, we detail each of these four steps.

Step 1: Constructing the PDTS. In this step, the PDTS $\mathcal{T}^{\times} = (L^{\times}, V^{\times}, l^{\times}_{\text{init}}, \theta^{\times}_{\text{init}}, \mapsto^{\times})$ is constructed from the given MDP $\mathcal{M}$ and the NBA N^{φ}, as explained in Section 4.1.

Step 2: Fixing templates. The algorithms fix a template for the affine distributional certificate $(\mathcal{C}, I)$, while the synthesis algorithm also fixes a template for the affine distributionally memoryless strategy π. The novelty, compared to prior work on verification and synthesis for distributional reachability and safety specifications [4, 5], lies in a more complex template design for the distributional certificate, which is now defined with respect to the PDTS:

- **Template for $\mathcal{C}$.** Recall that an affine distributional Büchi ranking function is of the form $\mathcal{C}(q,\mu) = \sum_{i=1}^{|S|} a_i^q \cdot \mu(s_i) + b^q$ as in eq. (3). Hence, the template for $\mathcal{C}$ is defined by introducing a set of symbolic template variables $a_1^q, \ldots, a_{|S|}^q, b^q$ for each NBA state $q \in Q$.
- **Template for I.** Similarly, the template for an affine distributional invariant I is of the form as in eq. (4). Hence, the template for I is defined by introducing a set of symbolic template variables $c_1^{k,q}, \ldots, c_{|S|}^{k,q}, d^{k,q}$ for each NBA state $q \in Q$ and each $k \in \{1, \ldots, N_I\}$, where N_I is the algorithm parameter that specifies the size of the invariant.
- **Template for π (synthesis algorithm).** The template for an affine distributionally memoryless strategy π is of the form as in eq. (5), hence it is defined by introducing symbolic template variables $e_{1,s,a}^{\pi}, \ldots, e_{|S|,s,a}^{\pi}, f_{s,a}^{\pi}$ and $g_{1,s}^{\pi}, \ldots, g_{|S|,s}^{\pi}, h_s^{\pi}$ for each state $s \in S$ and action $a \in Act$. Note that, in the special case when we are interested in synthesizing memoryless strategies instead of distributionally memoryless strategies, the strategy template becomes simpler. Instead of the template as in eq. (5), we introduce a single symbolic template variable $p_{s,a}^{\pi}$ for each state-action pair $s \in S$ and $a \in Act$, to encode the probability of taking action a in state s.

Step 3: Collecting constraints. In this step, the algorithms collect a system of constraints over the symbolic template variables that together encode that $\mathcal{C}$ and $\mathcal{I}$ indeed define a valid distributional certificate. For the synthesis algorithm, we also collect a system of constraints that encode that π defines a valid distributionally memoryless strategy. In each of the following constraints, each appearance of $\mathcal{C}$, $\mathcal{I}$ and π is replaced by the symbolic template introduced in Step 2, in the form as in eq. (3), (4) and (5). Moreover, we write $\mu \in \Delta(S)$ for the conjunction of affine inequalities $\wedge_{i=1}^{|S|}(\mu_i \geq 0) \wedge (\mu_1 + \cdots + \mu_{|S|} = 1)$.

- **Initial condition.** We define

$$\Phi_{\text{init}} \equiv \forall \mu \in \mathbb{R}^{|S|}.\, \mu \in \Delta(S) \wedge \mu \in \mathsf{Init} \implies \mathcal{C}(q_0,\mu) \geq 0 \wedge \bigwedge_{k=1}^{N_I} I^k(q_0,\mu) \geq 0.$$

- **Büchi ranking condition for accepting states.** For each accepting state $q \in F$ and letter $\sigma \in 2^{\mathsf{AP}}$ in the NBA, we define

$$\Phi_{\text{Büchi},q,\sigma} \equiv \forall \mu \in \mathbb{R}^{|S|}.\, \bigvee_{q' \in \delta(q,\sigma)} \mu \in \Delta(S) \wedge \mu \models G(\sigma) \wedge \mathcal{C}(q,\mu) \geq 0 \wedge \bigwedge_{k=1}^{N_I} I^k(q,\mu) \geq 0$$
$$\implies \mathcal{C}(q', \mathcal{M}^{\pi}(\mu)) \geq 0 \wedge \bigwedge_{k=1}^{N_I} I^k(\mathcal{M}^{\pi}(q',\mu)) \geq 0.$$

- **Büchi ranking condition for nonaccepting states.** For each non-accepting state $q \in Q \backslash F$ and letter $\sigma \in 2^{\mathsf{AP}}$ in the NBA in the NBA, we define

$$\Phi_{\text{Büchi},q,\sigma} \equiv \forall \mu \in \mathbb{R}^{|S|}.\, \bigvee_{q' \in \delta(q,\sigma)} \mu \in \Delta(S) \wedge \mu \models G(\sigma) \wedge \mathcal{C}(q,\mu) \geq 0 \wedge \bigwedge_{k=1}^{N_I} I^k(q,\mu) \geq 0$$
$$\implies \mathcal{C}(\mu,q) - 1 \geq \mathcal{C}(\mathcal{M}^{\pi}(\mu), q') \geq 0 \wedge \bigwedge_{k=1}^{N_I} I^k(\mathcal{M}^{\pi}(q',\mu)) \geq 0.$$

- **Strategy conditions (synthesis algorithm).** For the strategy template to indeed define a valid affine distributionally memoryless strategy, we require that:

$$\Phi_\pi \equiv \bigwedge_{s \in S} \Big(\sum_{a \in Act} \pi(s,a)(\mu) = 1 \wedge \bigwedge_{a \in Act} (\pi(s,a)(\mu) \geq 0) \Big).$$

In the above definitions, note that $\mathcal{M}^\pi(\mu)$ is the one-step successor from distribution μ when policy π is applied in the MDP, computed as: $\sum_{s \in S, a \in Act(s)} \pi(s,a)(\mu) \cdot P(s,a)$.

Step 4: Constraint solving. The strategy condition constraint Φ_π is a purely existentially quantified Boolean combination of affine inequalities over the symbolic template variables. However, constraints Φ_{init} and $\Phi_{\text{Büchi},q,\sigma}$ are all of the form

$$\forall \mu \in \mathbb{R}^{|S|}. \bigvee_{i=1}^{m} \bigwedge_{j=1}^{n} \text{aff-expr}_{i,j}(t,\mu) \geq 0 \geq \bigwedge_{l=1}^{k} \text{poly-expr}_k(t,\mu) \geq 0,$$

where t is the vector of all symbolic template variables, $\text{aff-expr}_{i,j}(t,\mu)$'s are some affine functions and poly-expr_k's are some polynomial functions over the vectors of variables t and μ. Polynomial expressions on the right hand side arise due to the quotients of affine expressions that define affine distributionally memoryless strategies, see eq. (5). Hence, multiplying both sides of the inequality by the affine expressions appearing in denominators results in polynomial expressions over variables in t and μ.

The problem of synthesizing affine distributional certificates and affine distributionally memoryless strategies then reduces to solving a system of constraints that contain quantifier alternation $\exists t. \forall \mu$. Such quantifier alternation over real-valued variables is generally hard to handle directly and can lead to inefficiency in solvers. In order to allow for a more efficient constraint solving, before passing our system of constraints to an SMT-solver, we first apply Handelman's theorem [27] to translate Φ_{init} and $\Phi_{\text{Büchi},q,\sigma}$ into a purely existentially quantified system of polynomial constraints over the symbolic template variables in t and auxiliary variables introduced by the translation, whose satisfiability implies satisfiability of the original constraints. This translation is common in template-based program analysis, see [7] for details. This step allows for more efficient constraint solving as well as better bound on the algorithm complexity. Finally, the resulting purely existentially quantified system of polynomial constraints over real-valued variables is solved via an SMT solver.

In the special case when we are interested in synthesizing memoryless strategies rather than distributionally memoryless strategies, we may use Farkas' lemma [25] rather than Handelman's theorem. This yields a *sound and complete* translation into an equisatisfiable purely existentially satisfied system of constraints.

Soundness, relative completeness, complexity. Soundness of our algorithms follows from the soundness of all four steps above, including soundness of the transformations via Handelman's theorem and Farkas' lemma [7]. Since the Farkas' lemma transformation leads to an equisatisfiable system of constraints, it also follows that our algorithm is *relatively complete* – it is guaranteed to synthesize an affine distributional certificate and memoryless strategy whenever they exist. Finally, our algorithms provide a PSPACE complexity upper bound as they reduce the synthesis and verification problems to solving a sentence in the existential first-order theory of the reals. The following theorem summarizes these results.

▶ **Theorem 12.**

Soundness: *If the algorithm returns an affine distributional certificate* $(\mathcal{C}, I)$ *and an affine distributionally memoryless strategy* π *(for the synthesis algorithm), then the MDP* $\mathcal{M}$ *with initial distributions* Init *under strategy* π *satisfies specification* φ*.*

Relative completeness: *If there exist an affine distributional certificate* $(\mathcal{C}, I)$ *and a memoryless strategy* π*, then there exists an invariant size* $N_I \in \mathbb{N}$ *such that* $(\mathcal{C}, I)$ *and* π *are computed by the algorithm.*

Complexity: *The runtime of the algorithm is in PSPACE in the size of the encoding of the MDP, NBA* N^{φ}*, startegy* π *(for the verification algorithm) and invariant size* $N_I \in \mathbb{N}$*.*

6 Experimental Evaluation

We implemented a prototype of our method in Python 3 and experimentally evaluated it on a number of challenging verification and synthesis tasks collected from the literature on distributional specifications in MDPs. Our prototype takes as input an MDP (in the Prism [30] input format) and an LTL specification. The LTL specification is then translated into an NBA via Spot [24]. For the constraint solving step in our algorithms, we use PolyQEnt [16] which provides a tooling support for quantifier elimination via Farkas' lemma and Handelman's theorem. PolyQEnt uses Z3 [23] and MathSAT5 [20] as backend SMT solvers for the final system of purely existentially quantified constraints. We set the invariant size parameter to $N_I = 1$, which was sufficient for all our experiments. Our experiments were conducted on consumer-grade hardware (AMD Ryzen 5 5625U CPU, 8GB RAM).

Benchmarks. We evaluated our method on several examples collected from the literature:

- **GridWorld (synthesis).** Motivated by [5], these benchmarks model robot swarms in gridworld environments. Initially, all robots are placed in the top-left corner of the gridworld environment. Some of the cells are covered by walls whereas some are slippery and with certain probability may lead to moving in an undesired direction. Hence, each environment induces an MDP. As shown in [5], the evolution of a robot swarm can be analyzed by taking the distribution transformer view of MDPs and considering how the robots are distributed across the gridworld cells at each time step. In Table 1, we consider 5 gridworld benchmarks of varying sizes and consider two distributional specifications: (1) at least 90% of robots should be in some slippery target cell infinitely often, and (2) in addition, at most 50% of robots should occupy some narrow passage at any point in time.
- **PageRank (verification).** We consider a Markov chain representation of the PageRank algorithm taken from [2]. Given the context, we consider various verification tasks, which are of the form: always if the probability mass at some vertex/page is above a threshold, then eventually, it must be above a threhold in another page.
- **Pharamakocinetics (verification)** We also consider a 6 state Markov chain from [2] which is adapted from a Pharmacokinetics example in [11]. We use the two queries that were listed in [2] as the motivating examples to obtain our specifications.
- **Benchmarks from [4] (verification and synthesis)**. Finally, we collect 3 pairs of verification and synthesis tasks from [4]. In the verification task a strategy is fixed, whereas in the synthesis task one also needs to compute the strategy. While [4] considered distributional safety specifications, we design more complex ω-regular specifications.

Results. Our experimental results are shown in Table 1. Our results demonstrate that our prototype is able to solve a number of challenging verification and synthesis tasks for distributional ω-regular specifications in MDPs, that were beyond the reach of all existing

Table 1 For each experiment we report, from left to right, the benchmark, specification, task (verification or synthesis), the number of coefficients, the number of constraints, the number of coefficients in PolyQEnt generated file (i.e. after application of Farkas' lemma), the number of constraints in PolyQEnt generated file, SMT-solving time, and the total runtime.

Model	Specification	Task	Coeff #	Const #	PQ Coeff #	Query #	SMT time	Total time
GW (3*3)	G F "V5>=0.9"	Synth	62	41	171	34	$< 2s$	$6s$
GW (3*3)	G F "V5>=0.9" & G "V4<=0.5"	Synth	80	45	277	49	$< 5s$	$< 5s$
GW (4*4)	G F "V11>=0.9"	Synth	121	79	286	81	$< 10s$	$10s$
GW (4*4)	G F "V11>=0.9" & G "V9<=0.5"	Synth	153	83	448	87	$12s$	$13s$
GW (5*5)	G F "V19>=0.9"	Synth	198	129	435	131	$302s$	$303s$
PageRank	F G "V2>0.2"	Verify	60	13	400	35	$63s$	$64s$
PageRank	G ("0.2<=V0" → F "0.2<=V2")	Verify	48	13	253	22	$17s$	$18s$
PageRank	G ("0.2<=V2" → F "0.2<=V2")	Verify	48	13	253	22	$8s$	$8s$
PageRank	G ("0.2<=V3" → F "0.2<=V2")	Verify	48	13	253	22	$44s$	$45s$
PageRank	G ("0.2<=V4" → F "0.2<=V2")	Verify	48	13	253	22	$5s$	$6s$
PageRank	G "V0>=0.2" | "V1>=0.2" | "V2>=0.2" | "V3>=0.2" | "V4>=0.2" → F "V2=1"	Verify	60	19	569	44	$136s$	$137s$
PageRank	F "V2=1"→ G "V1<=0.2"	Verify	144	35	630	46	$6s$	$6s$
Pharmacokinetics	F "V4=1"	Verify	42	9	176	14	$< 1s$	$< 1s$
Pharmacokinetics	G ("0.13<=V3<=0.2" | "0<=V3<0.13")	Verify	56	11	365	28	$102s$	$102s$
CAV23 [4]	G F "V1>=0.249"	Verify	16	7	85	13	$< 2s$	$< 2s$
CAV23 [4]	G F "V1>=0.249"	Synth	20	14	108	16	$7s$	$< 2s$
CAV23 [4]	"V1>= 0.249" U "V2 >= 0.25"	Verify	32	13	185	22	$< 5s$	$7s$
CAV23 [4]	"V1>= 0.249" U "V2 >= 0.25"	Synth	36	20	189	25	$7s$	$5s$
CAV23 [4]	"0.334>=V1>=0.332" U "V0=0.25"	Verify	32	17	229	24	$< 4s$	$4s$
CAV23 [4]	"0.334>=V1>=0.332" U "V0=0.25"	Synth	36	20	233	28	$< 1s$	$< 1s$

methods. This is achieved at runtimes that are comparable or even lower than those reported by earlier methods for distributional reachability and safety specifications in [4, 5] on benchmarks of similar size. Hence, even though we consider a significantly more general class of distributional ω-regular specifications, our algorithms do not lead to a significant computational overhead. Moreover, for all our strategy synthesis tasks, our prototype was able to compute *memoryless strategies* that lead to distributional specification satisfaction. This demonstrates the generality of relative completeness guarantees provided by our algorithms.

We also make some observations. First, as can be seen from runtimes reported in Table 1, the final SMT-solving step is computationally the most expensive step of our algorithms. Constraint generation took at most a few seconds in all cases. Second, we observe that strategy synthesis tasks are generally computationally more expensive, which is expected given that they require synthesizing both the strategy and the distributional certificate. However, in some cases the synthesis problems can also be solved more efficiently. This is demonstrated by the last 6 experiments (CAV'23 in Table 1), where synthesis is achieved at lower runtimes due to our prototype being able to compute a simple memoryless strategy that was easier to verify, compared to the strategy considered in the verification task.

Finally, regarding the invariant size parameter, we used $N_I = 1$ because it was sufficient for all our benchmarks. We also ran our prototype tool with $N_I = 2$ on 12 of the benchmarks and 8 of them were solved within the timeout of 5 minutes. The timeouts are likely due to the larger size of the final system of constraints. Indeed, given more time, we expect our method can be effectively applied with larger template sizes as well.

7 Conclusion

In this paper, we considered distributional ω-regular specifications in MDPs and addressed the problems of strategy verification and synthesis. We developed new notions of product distributional transition systems between an MDP and an NBA. We then introduced distributional certificates, using which we provided template-based synthesis algorithms for

strategy verification and synthesis. Our experiments demonstrate the benefits and promise of our approach. As future work, we would like to go beyond MDPs and consider partially observable MDPs. Moreover, it would be interesting to lift the objectives from NBA to Rabin automata, where even the notion of distributional certificates is unclear.

References

1 Alessandro Abate, Mirco Giacobbe, and Diptarko Roy. Stochastic omega-regular verification and control with supermartingales. In Arie Gurfinkel and Vijay Ganesh, editors, *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part III*, volume 14683 of *Lecture Notes in Computer Science*, pages 395–419. Springer, 2024. doi:10.1007/978-3-031-65633-0_18.

2 Manindra Agrawal, S. Akshay, Blaise Genest, and P. S. Thiagarajan. Approximate verification of the symbolic dynamics of Markov chains. *J. ACM*, 62(1):2:1–2:34, 2015. doi:10.1145/2629417.

3 S. Akshay, Timos Antonopoulos, Joël Ouaknine, and James Worrell. Reachability problems for Markov chains. *Inf. Process. Lett.*, 115(2):155–158, 2015. doi:10.1016/J.IPL.2014.08.013.

4 S. Akshay, Krishnendu Chatterjee, Tobias Meggendorfer, and Dorde Zikelic. MDPs as distribution transformers: Affine invariant synthesis for safety objectives. In Constantin Enea and Akash Lal, editors, *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part III*, volume 13966 of *Lecture Notes in Computer Science*, pages 86–112. Springer, 2023. doi:10.1007/978-3-031-37709-9_5.

5 S. Akshay, Krishnendu Chatterjee, Tobias Meggendorfer, and Dorde Zikelic. Certified policy verification and synthesis for MDPs under distributional reach-avoidance properties. In *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence, IJCAI 2024, Jeju, South Korea, August 3-9, 2024*, pages 3–12. ijcai.org, 2024. URL: https://www.ijcai.org/proceedings/2024/1.

6 S. Akshay, Blaise Genest, and Nikhil Vyas. Distribution-based objectives for Markov decision processes. In Anuj Dawar and Erich Grädel, editors, *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 36–45. ACM, 2018. doi:10.1145/3209108.3209185.

7 Ali Asadi, Krishnendu Chatterjee, Hongfei Fu, Amir Kafshdar Goharshady, and Mohammad Mahdavi. Polynomial reachability witnesses via stellensätze. In Stephen N. Freund and Eran Yahav, editors, *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, pages 772–787. ACM, 2021. doi:10.1145/3453483.3454076.

8 Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.

9 Roberto Baldoni, François Bonnet, Alessia Milani, and Michel Raynal. On the solvability of anonymous partial grids exploration by mobile robots. In Theodore P. Baker, Alain Bui, and Sébastien Tixeuil, editors, *Principles of Distributed Systems, 12th International Conference, OPODIS 2008, Luxor, Egypt, December 15-18, 2008. Proceedings*, volume 5401 of *Lecture Notes in Computer Science*, pages 428–445. Springer, 2008. doi:10.1007/978-3-540-92221-6_27.

10 Danièle Beauquier, Alexander Moshe Rabinovich, and Anatol Slissenko. A logic of probability with decidable model checking. *J. Log. Comput.*, 16(4):461–487, 2006. doi:10.1093/logcom/exl004.

11 Rohit Chadha, Vijay Anand Korthikanti, Mahesh Viswanathan, Gul Agha, and YoungMin Kwon. Model checking MDPs with a unique compact invariant set of distributions. In *Eighth International Conference on Quantitative Evaluation of Systems, QEST 2011, Aachen, Germany, 5-8 September, 2011*, pages 121–130. IEEE Computer Society, 2011. doi:10.1109/QEST.2011.22.

12 Aleksandar Chakarov and Sriram Sankaranarayanan. Probabilistic program analysis with martingales. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*

- *25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 511–526. Springer, 2013. doi:10.1007/978-3-642-39799-8_34.

13 Krishnendu Chatterjee, Hongfei Fu, and Amir Kafshdar Goharshady. Termination analysis of probabilistic programs through positivstellensatz's. In *CAV (1)*, volume 9779 of *Lecture Notes in Computer Science*, pages 3–22. Springer, 2016. doi:10.1007/978-3-319-41528-4_1.

14 Krishnendu Chatterjee, Hongfei Fu, Amir Kafshdar Goharshady, and Ehsan Kafshdar Goharshady. Polynomial invariant generation for non-deterministic recursive programs. In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 672–687. ACM, 2020. doi:10.1145/3385412.3385969.

15 Krishnendu Chatterjee, Hongfei Fu, Petr Novotný, and Rouzbeh Hasheminezhad. Algorithmic analysis of qualitative and quantitative termination problems for affine probabilistic programs. *TOPLAS*, 40(2):7:1–7:45, 2018. doi:10.1145/3174800.

16 Krishnendu Chatterjee, Amir Kafshdar Goharshady, Ehsan Kafshdar Goharshady, Mehrdad Karrabi, Milad Saadat, Maximilian Seeliger, and Đorđe Žikelić. Polyqent: A polynomial quantified entailment solver, 2025. arXiv:2408.03796.

17 Krishnendu Chatterjee, Amir Kafshdar Goharshady, Ehsan Kafshdar Goharshady, Mehrdad Karrabi, and Dorde Zikelic. Sound and complete witnesses for template-based verification of LTL properties on polynomial programs. In André Platzer, Kristin Yvonne Rozier, Matteo Pradella, and Matteo Rossi, editors, *Formal Methods - 26th International Symposium, FM 2024, Milan, Italy, September 9-13, 2024, Proceedings, Part I*, volume 14933 of *Lecture Notes in Computer Science*, pages 600–619. Springer, 2024. doi:10.1007/978-3-031-71162-6_31.

18 Krishnendu Chatterjee, Amir Kafshdar Goharshady, Tobias Meggendorfer, and Dorde Zikelic. Sound and complete certificates for quantitative termination analysis of probabilistic programs. In *CAV (1)*, volume 13371 of *Lecture Notes in Computer Science*, pages 55–78. Springer, 2022. doi:10.1007/978-3-031-13185-1_4.

19 Krishnendu Chatterjee, Petr Novotný, and Đorđe Žikelić. Stochastic invariants for probabilistic termination. In *POPL*, pages 145–160, 2017. doi:10.1145/3009837.3009873.

20 Alessandro Cimatti, Alberto Griggio, Bastiaan Schaafsma, and Roberto Sebastiani. The MathSAT5 SMT Solver. In Nir Piterman and Scott Smolka, editors, *Proceedings of TACAS*, volume 7795 of *LNCS*. Springer, 2013. doi:10.1007/978-3-642-36742-7_7.

21 Michael Colón, Sriram Sankaranarayanan, and Henny Sipma. Linear invariant generation using non-linear constraint solving. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*, volume 2725 of *Lecture Notes in Computer Science*, pages 420–432. Springer, 2003. doi:10.1007/978-3-540-45069-6_39.

22 Michael Colón and Henny Sipma. Synthesis of linear ranking functions. In Tiziana Margaria and Wang Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 7th International Conference, TACAS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*, volume 2031 of *Lecture Notes in Computer Science*, pages 67–81. Springer, 2001. doi:10.1007/3-540-45319-9_6.

23 Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: an efficient SMT solver. In *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008. doi:10.1007/978-3-540-78800-3_24.

24 Alexandre Duret-Lutz, Etienne Renault, Maximilien Colange, Florian Renkin, Alexandre Gbaguidi Aisse, Philipp Schlehuber-Caissier, Thomas Medioni, Antoine Martin, Jérôme Dubois, Clément Gillard, et al. From spot 2.0 to spot 2.10: What's new? In *International Conference on Computer Aided Verification*, pages 174–187. Springer, 2022.

25 Julius Farkas. Theorie der einfachen ungleichungen. *Journal für die reine und angewandte Mathematik (Crelles Journal)*, 1902(124):1–27, 1902.

26 Yulong Gao, Alessandro Abate, Lihua Xie, and Karl Henrik Johansson. Distributional reachability for Markov decision processes: Theory and applications. *IEEE Trans. Autom. Control.*, 69(7):4598–4613, 2024. doi:10.1109/TAC.2023.3341282.

27 David Handelman. Representing polynomials by positive linear functions on compact convex polyhedra. *Pacific Journal of Mathematics*, 132(1):35–62, 1988.

28 Thomas A. Henzinger, Maria Mateescu, and Verena Wolf. Sliding window abstraction for infinite Markov chains. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, volume 5643 of *Lecture Notes in Computer Science*, pages 337–352. Springer, 2009. doi:10.1007/978-3-642-02658-4_27.

29 Vijay Anand Korthikanti, Mahesh Viswanathan, Gul Agha, and YoungMin Kwon. Reasoning about MDPs as transformers of probability distributions. In *QEST 2010, Seventh International Conference on the Quantitative Evaluation of Systems, Williamsburg, Virginia, USA, 15-18 September 2010*, pages 199–208. IEEE Computer Society, 2010. doi:10.1109/QEST.2010.35.

30 Marta Z. Kwiatkowska, Gethin Norman, and David Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *CAV*, volume 6806 of *Lecture Notes in Computer Science*, pages 585–591. Springer, 2011. doi:10.1007/978-3-642-22110-1_47.

31 YoungMin Kwon and Gul A. Agha. Verifying the evolution of probability distributions governed by a DTMC. *IEEE Trans. Software Eng.*, 37(1):126–141, 2011. doi:10.1109/TSE.2010.80.

32 Joël Ouaknine and James Worrell. Decision problems for linear recurrence sequences. In Alain Finkel, Jérôme Leroux, and Igor Potapov, editors, *Reachability Problems - 6th International Workshop, RP 2012, Bordeaux, France, September 17-19, 2012. Proceedings*, volume 7550 of *Lecture Notes in Computer Science*, pages 21–28. Springer, 2012. doi:10.1007/978-3-642-33512-9_3.

33 Stephen Prajna, Ali Jadbabaie, and George J. Pappas. A framework for worst-case and stochastic safety verification using barrier certificates. *IEEE Trans. Autom. Control.*, 52(8):1415–1428, 2007. doi:10.1109/TAC.2007.902736.

34 Toru Takisaka, Yuichiro Oyabu, Natsuki Urabe, and Ichiro Hasuo. Ranking and repulsing supermartingales for reachability in randomized programs. *ACM Trans. Program. Lang. Syst.*, 43(2):5:1–5:46, 2021. doi:10.1145/3450967.

35 Dorde Zikelic, Mathias Lechner, Thomas A. Henzinger, and Krishnendu Chatterjee. Learning control policies for stochastic systems with reach-avoid guarantees. In Brian Williams, Yiling Chen, and Jennifer Neville, editors, *Thirty-Seventh AAAI Conference on Artificial Intelligence, AAAI 2023, Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence, IAAI 2023, Thirteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2023, Washington, DC, USA, February 7-14, 2023*, pages 11926–11935. AAAI Press, 2023. doi:10.1609/AAAI.V37I10.26407.

Temporal Explorability Games

Pete Austin
University of Liverpool, UK

Sougata Bose
UMONS – Université de Mons, Belgium

Nicolas Mazzocchi
Slovak University of Technology in Bratislava, Slovakia

Patrick Totzke
University of Liverpool, UK

Abstract

Temporal graphs extend ordinary graphs with discrete time that affects the availability of edges. We consider solving games played on temporal graphs where one player aims to explore the graph, i.e., visit all vertices. The complexity depends majorly on two factors: the presence of an adversary and how edge availability is specified.

We demonstrate that on static graphs, where edges are always available, solving explorability games is just as hard as solving reachability games. In contrast, on temporal graphs, the complexity of explorability coincides with generalized reachability (NP-complete for one-player and PSPACE-complete for two player games). We show that if temporal graphs are given symbolically, even one-player reachability (and thus explorability and generalized reachability) games are PSPACE-hard. For one player, all these are also solvable in PSPACE and for two players, they are in PSPACE, EXP and EXP, respectively.

2012 ACM Subject Classification Theory of computation → Network games; Theory of computation → Representations of games and their complexity; Mathematics of computing → Graphs and surfaces; Theory of computation → Formal languages and automata theory

Keywords and phrases Temporal Graphs, Explorability, Reachability, Games

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2025.7

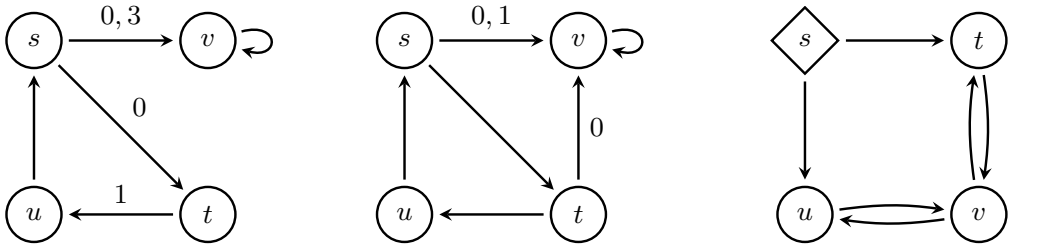
Funding This work has been supported by the EPSRC, projects EP/V025848/1 and EP/X042596/1 and by the Fonds de la Recherche Scientifique – FNRS under Grant n° T.0188.23 (PDR ControlleRS). The research was mainly performed when N. Mazzocchi was affiliated with the Institute of Science and Technology Austria where he was financed by the ERC-2020-AdG 101020093.

1 Introduction

Two player zero-sum games on graphs are a common formalism in formal verification, especially reactive synthesis (see e.g. [21, 16, 34, 35]). The two players jointly determine an infinite path stepwise, where the owner of the current vertex gets to extend the path to a valid successor. One player aims to satisfy a given winning condition, such as reaching a target vertex, and the opposing player aims to prevent that. Under very general conditions, that are satisfied here, such games are determined, meaning that exactly one player has a winning strategy that guarantees a favourable outcome for them no matter their opponent's choices. Solving a game refers to the algorithmic task to determine which player has a winning strategy.

Temporal graphs are a way to model dynamic systems: they extend graphs with discrete and global time and can specify for each edge at which times it can be traversed. Temporal graphs are typically given as sequence of graphs over the same vertex set, an encoding we refer to as *explicit*. For our purposes this is polynomially equivalent to encoding temporal

© Pete Austin, Sougata Bose, Nicolas Mazzocchi, and Patrick Totzke;
licensed under Creative Commons License CC-BY 4.0



(a) Arena $\mathcal{A}_1$: explorable from s, not from t, u, v.

(b) Arena $\mathcal{A}_2$: not explorable from any vertex.

(c) Arena $\mathcal{A}_3$: explorable from s but the order is not enforceable.

Figure 1 Examples of games on temporal graphs. Vertices owned by Player 1 are drawn as circles, those owned by Player 2 as diamonds. The labels on edges denote the times at which they are available. Edges without labels are always available.

graphs as ordinary graphs in which (directed) edges carry an explicit list of timestamps, which we also refer to as explicit encodings[1]. Alternatively, and more interestingly, we consider succinct symbolic representations where the availability of edges is given as logical predicate in the existential fragment of Presburger Arithmetic.

In this paper we study the computational complexity of solving games played on temporal graphs (where players' choices are required to respect the temporal constraints on edges). Our focus lies on *explorability* games, where Player 1 wins if and only if all vertices of the graph are visited. Explorability generalizes reachability conditions in the sense that reachability games straightforwardly and in logarithmic space reduce to explorability games. On the other hand, explorability is a special case of so-called *generalized reachability* conditions of [17], where several target sets are given and at least one vertex must be reached in each.

For the benefit of readers unfamiliar with temporal graphs, or turn-based games, we start by providing some intuition to help appreciate the difficulties caused by the dynamic changes of the arena, as well as the presence of antagonistic choice. Consider first the one-player explorability game played on the arena $\mathcal{A}_1$ in Figure 1a. Starting in s, Player 1 wins by exploring the graph as $s \to t \to u \to v$. From any other vertex there is no exploring path due to the non-availability of edges at time 0. An easy property that implies non-explorability of static graphs is the presence of two pairwise non-reachable vertices. Indeed, if two such vertices exists then any path can at most see one of them, hence not explore the graph. This can be turned into a full characterization of explorability even in the presence of an opponent (see Lemma 4). However, due to the temporal constraints on edges, this cannot naïvely be extended to temporal graphs. For example, arena $\mathcal{A}_1$ is explorable yet v is not reachable from t starting at time 0, and vice versa. On the other hand, in arena $\mathcal{A}_2$ (Figure 1b), for every pair of vertices at least one can reach the other starting at time 0. Yet, from no initial vertex and at no time, $\mathcal{A}_2$ is explorable.

Figure 1c demonstrates that even if Player 1 wins the explorability game (as here from vertex s), she may not be able to enforce visiting the vertices in a particular order, as that can be influenced by her opponent's moves. Dually, as for the game starting in vertex s in $\mathcal{A}_2$, even if Player 2 wins, he may not be able to dictate which vertex is left unexplored.

[1] This is trivial for timestamps in unary encoding; Allowing binary-encoded timestamps in the input does not impact the complexity of solving explorability games since winning plays necessarily use consecutive times and thus cannot exist if the edge relation is too sparse.

Related Work. Temporal graphs have been used to analyse dynamic networks and distributed systems in dynamic topologies, such as dissemination/propagation of information [9, 12, 24, 36] or the spread of diseases [38]. There is a large body of mathematical work that considers temporal generalizations of various graph-theoretic notions and properties [11, 18, 31]. Related algorithmic questions include graph colouring [28], travelling salesman [32], maximal matchings [27], and checking the existence of temporal cliques [29], Eulerian circuits [7], vertex-cover sets [3], and explorability [1, 2, 14] (called exploration). Several prior works on explorability assume structural properties that ensure that the graph is explorable. Questions then focus on the minimal time to explore. Pelc [33] provides several sufficient conditions on temporal graphs to be explorable. Spirakis and Michail [32] showed that without assumptions on the graph, checking explorability can be done in linear time for static graphs and is NP-complete for temporal graphs. In most of the works, a path in the temporal graph is allowed to wait at a vertex for an unbounded amount of time.

The edge relation is often deliberately left unspecified and sometimes only assumed to satisfy some weak assumptions about connectedness, frequency, or fairness to study the worst or average cases in uncontrollable environments. Depending on the application, one distinguishes between "online" questions (e.g. [19, 26]), where the edge availability is revealed stepwise, as opposed to the "offline" variant where all is given in advance. We refer to [13, 22, 30] for overviews of temporal graph theory and its applications.

Fijalkow and Horn [17] study games with *generalized reachability* objectives, which generalize not only reachability but also explorability games. Generalized reachability conditions are conjunctions of reachability conditions: *Player* 1 aims to reach at least one vertex out of each of several target sets. Explorability thus corresponds to the case where every vertex forms a singleton target set. Solving generalized reachability games is PSPACE-complete [17], but in polynomial time if all target sets are singletons.

Reachability and parity games played on (symbolically represented) temporal graphs have been introduced in [6]; solving these games is PSPACE-complete. The lower bound was shown by reduction from the emptiness problem for unary alternating finite automata, crucially relying on the presence of antagonistic choice. As in [6], our notion of temporal graphs assumes that some edge must be traversed at every unit time. Waiting (not moving the token for a while) as occasionally permitted [30, 3] can be modelled by adding self-loops.

Turn-based games involving temporal constraints have also been studied in the context of games played on the configuration graphs of timed automata [4]. Solving timed parity games is complete for EXP [8, 25] and the lower bound already holds for reachability games on timed automata with only two clocks [23]. However, the time in temporal graphs is discrete as opposed to the continuous time semantics in timed automata. A direct translation of (games on) temporal graphs to equivalent timed automata games requires two clocks: one to hold the global time used to check the edge predicate and one to ensure that time progresses one unit per step. Fearnley and Jurdziński [15] showed that the reachability problem (solving one-player reachability games) for two-clock timed automata is already PSPACE-hard. Our lower bound constructions have a similar flavour but are incomparable in that they do not re-prove nor are implied by their result. They make crucial use of resetting of clocks which is impossible in our model. Our construction in turn uses either antagonistic choice (Theorem 8) or the more powerful transition guards in symbolic representations (Theorem 9).

Contributions. We study the complexity of solving explorability games and contrast the worst-case complexity for related decision questions for games played on static graphs versus temporal graphs. It turns out that explorability is no harder than reachability on static

		Static	Explicit	Symbolic
One-player	**Reachability**	NL-complete [5, Theorem 4.18]	NL-complete [Theorem 6]	**PSPACE-complete** [Corollary 14]
	Explorability	**NL-complete** [Theorem 5]	**NP-complete** [Theorem 7]	**PSPACE-complete** [Corollary 14]
	Gen. Reach	NP-complete [17, Theorem 3]	NP-complete [Theorem 6]	**PSPACE-complete** [Corollary 14]
Two-player	**Reachability**	P-complete [21]	P-complete [Theorem 6]	PSPACE-complete [6, Theorem 2]
	Explorability	**P-complete** [Theorem 5]	**PSPACE-complete** [Theorem 8]	**PSPACE-hard; In EXP** [Corollary 14]
	Gen. Reach	PSPACE-complete [17, Theorem 1]	PSPACE-complete [Theorem 6]	**PSPACE-hard; In EXP** [Corollary 14]

Figure 2 A table detailing the computational complexities for the one and two-player variants of reachability, explorability, and generalized reachability games played on static, explicitly and symbolically represented temporal graphs. New results are in boldface.

graphs whereas on temporal graphs, explorability games exhibit the hardness of generalized reachability when the temporal edge availability is given explicitly. For temporal explorability games with a succinct, symbolic representation, we have PSPACE-hardness for both variants but a PSPACE-EXP complexity gap for two-player games. Specifically,

1. on static graphs, solving explorability games is complete for polynomial time (and NL-complete for one-player games);
2. on explicitly represented temporal graphs, explorability games are PSPACE-complete (NP-complete in the one-player variant);
3. reachability and thus explorability games on symbolically represented temporal graphs are PSPACE-hard even in the single-player variant. This strengthens the known lower bound from [6] which required a second antagonistic player.

Figure 2 summarizes these and related claims. Our most involved constructions are the PSPACE lower bounds, both from the satisfiability of quantified Boolean formulae (QBF).

2 Notations

▶ **Definition 1** (Temporal Graphs). *A temporal graph $G = (V, E)$ is a directed graph where V are vertices and $E : V^2 \to 2^{\mathbb{N}}$ is the edge availability relation that maps each pair of vertices to the set of times at which the respective directed edge can be traversed. If $i \in E(u, v)$ we call v an i-*successor *of u and write $u \xrightarrow{i} v$. We call G* static *if for all $u, v \in V$, $E(u, v)$ is either $\mathbb{N}$ or $\emptyset$. Its* horizon *is $h(G) = \sup_{u,v \in V}(E(u, v))$, that is, the largest finite time at which any edge is available, or ∞ if no such finite time exists.*

Naturally, one can unfold a temporal graph G into its *expansion* up to time $T \in \mathbb{N} \cup \{\infty\}$, which is the static graph G_T with vertices $V \times \{0, 1, \ldots, T, T + 1\}$ and $(u, \theta) \to (v, \theta + 1)$ if and only if $\theta \in E(u, v)$. We denote by *the expansion* of a temporal graph G, its expansion up to its horizon $h(G)$. See for instance Figure 3 for the expansion of a temporal graph $\mathcal{A}_1$ (up to its horizon $h(\mathcal{A}_1) = 3$).

Complexity bounds for decision problems about temporal graphs very much depend on the representation of the input. We will say a temporal graph is represented *explicitly* if it is given as sequence of directed graphs, one per unit time from 0 up to its (finite) horizon.

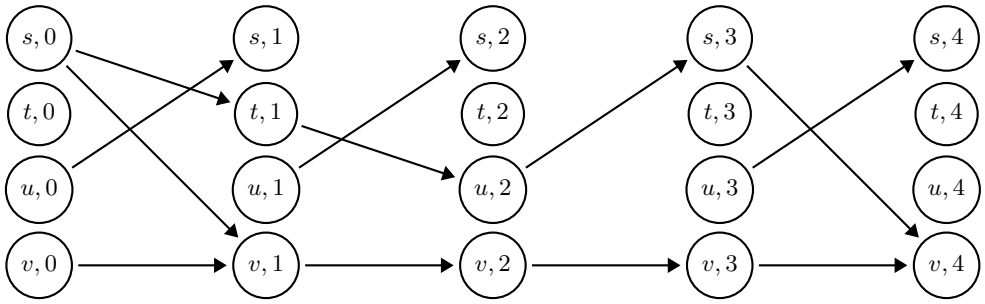


Figure 3 The expansion of $\mathcal{A}_1$.

We also consider a *symbolic* representation where the edge relation E is represented as a formula in the existential fragment of Presburger Arithmetic ($\exists$PA), the first-order theory over natural numbers with equality and addition. That is, the formula $\Phi_{u,v}(x)$ with one free variable x represents the set of times at which an edge from u to v is available as $E(u,v) = \{n \mid \Phi_{u,v}(n) \equiv true\}$. We use common syntactic sugar including inequality and multiplication with constants.

In this representation, checking if an edge is available at a given time, i.e., checking whether a given $\exists$PA formula is satisfied by a given valuation, is NP-complete (and in polynomial time if the number of quantifiers are fixed) [37, Corollary 1]. The symbolic representation of a temporal graph can be exponentially more succinct than the explicit one: using repeated doubling one can express exponentially large values. This representation is also at most exponentially larger because the Presburger-definable edge relation must be ultimately periodic with base and period at most exponential [20].

▶ **Definition 2** (Games on graphs). *A* game *is played by two opposing players. It consists of a directed graph (V, E), a partitioning $V = V_1 \uplus V_2$ of the vertices into those controlled by Player 1 and Player 2 respectively, and a* winning condition. *We refer to $\mathcal{A} = (V_1, V_2, E)$ as the* arena *of the game.*

The game starts with a token on an initial vertex $s_0 \in V$ and is turn-based, where in round j, the owner of the vertex occupied by the token moves it to some successor. This way an infinite path $\rho = s_0 s_1 \ldots$ called a play *is generated based on choices made by each player given the current round and vertex. A play is won by Player 1 if it satisfies the given winning condition, and by Player 2 otherwise.*

A strategy *for Player i is a recipe for how to move. It is a function $\sigma_i : V^* V_i \to V$ from finite paths ending in a vertex s in V_i to some successor. A strategy is* winning from $s \in V$ *if Player i wins every play that starts in s and during which all decisions are according to σ_i.*

We call a vertex s winning for Player i if there exists a winning strategy from s, and call the subset of all such vertices the *winning region* for that player. The main algorithmic question is to *solve* a game, meaning in other words to compute the winning regions for a given arena and winning condition. We consider the following winning conditions.

- *Reachability* towards a given set $F \subseteq V$ of target vertices. This is satisfied by those plays that eventually visit a vertex in F.
- *Generalized Reachability* towards target sets $F_1, F_2, \ldots, F_k \subseteq V$. This is satisfied by those plays that visit at least one vertex of every target set F_i.
- *Explorability* is the condition that asks that eventually every vertex in V is visited.

Note that explorability is a special case of generalized reachability where every vertex $s_i \in V$ corresponds to one target set $F_i = \{s_i\}$.

We study games played on the expansion of a given temporal graph $G = (V, E)$, where the ownership of vertices and winning condition are defined on the underlying temporal graph G and lifted to *the expansion* of G. That is, for any time θ and $u \in V_i$, the vertex (u, θ) is owned by Player i. Similarly, (generalized) reachability conditions are defined in terms of target sets $F \subseteq V$ and explorability asks to visit every vertex in V.

3 Static Graphs

We first discuss the bounds of explorability games for both one and two player variants, on static graphs, i.e., every edge is available at any time.

▶ **Lemma 3.** *For every arena $\mathcal{A} = (V_1, V_2, E)$ and vertices $s, t \in V$ one can construct (in logarithmic space) an arena $\mathcal{B} = (V_1', V_2', E')$ such that $V' = V_1' \cup V_2'$, $V \subseteq V'$, $V_2 = V_2'$ and Player 1 wins the reachability game on $\mathcal{A}$ from s to t if, and only if, she wins the explorability game on $\mathcal{B}$.*

Proof. Without loss of generality, assume that $t \in V_1$ and that each vertex has at least one outgoing edge. We construct $\mathcal{B}$ as follows. Firstly, for every edge between vertices v, u, introduce a new vertex $[v, u] \in V_1'$ that can either move to u or reset, i.e. move back to s. Secondly, from target t there exists an edge to every other vertex.

We claim that Player 1 wins the reachability game on $\mathcal{A}$ if and only if she wins the explorability game on $\mathcal{B}$. Indeed, if Player 1 does not win on $\mathcal{A}$, then she cannot visit t on either arena and therefore loses in both. Conversely, if Player 1 does win on $\mathcal{A}$, then she can explore $\mathcal{B}$ by repeatedly moving from s to t; visit a previously unseen vertex, then reset to s. Note that a reduction from a one-player reachability game where Player 1 owns all the vertices will construct another one-player explorability game where this still holds. ◀

The following characterization of explorability games is adapted from [17, Theorem 4], where $s \preceq t$ denotes that Player 1 wins the reachability game from s with target t.

▶ **Lemma 4.** *Consider an arena with vertex set V and let $s \in V$ be an initial vertex. Player 1 wins the explorability game from s if, and only if, both 1) for all $u, v \in V$ either $u \preceq v$ or $v \preceq u$; and 2) for all $u \in V$, $s \preceq u$.*

Proof. Suppose both conditions 1 and 2 are true. Point 1 implies that there exists a linearization of all n vertices such that $v_1 \preceq v_2 \preceq \ldots \preceq v_n$. By Point 2 we have that $s \preceq v_1$ and therefore there is such a linearization of $\preceq$ with $v_1 = s$. Consequently, for all $1 \leq i < n$ there exists a Player 1 strategy σ_i that wins the reachability game from v_i to v_{i+1}. Player 1 can follow the σ_i from v_i until v_{i+1} is reached, then switch to σ_{i+1} and so on, until all vertices have been visited.

Suppose we have an arena where the first condition is false: there are $u, v \in V$ with $u \not\preceq v$ and $v \not\preceq u$. Regardless of the starting vertex s, once a play visits u, Player 1 cannot ensure to visit v from then on (or vice versa). Finally, if the second condition is false then there must be one vertex u that Player 1 cannot ensure to visit from s. ◀

▶ **Theorem 5.** *Solving one-player (respectively two-player) explorability games on a static graph is complete for* NL *(respectively* P*).*

Proof. The hardness is a consequence of Lemma 3. Note that the explorability game constructed from a one-player reachability game is also one-player. As one and two-player reachability games are NL-hard and P-hard respectively, the lower bounds follow. The upper

bounds follow from Lemma 4, observing that at most n^2 reachability queries are necessary to verify the two conditions in the lemma. These queries can be done in NL in the one-player case, and P in the two-player case. ◀

4 Explicitly represented Temporal Graphs

Before discussing explorability games, we first remark that solving reachability and generalized reachability games have the same complexity on explicitly represented temporal graphs and on static graphs. This is because for any temporal graph $\mathcal{A}$ one can construct its expansion $\mathcal{A}'$, which is only polynomially larger assuming explicit encodings, and modify the winning conditions appropriately to get a game on this static graph: every target vertex v in $\mathcal{A}$ gives rise to target vertices (v, θ) in $\mathcal{A}'$ for all times $0 \leq \theta \leq h(\mathcal{A})$.

This reduction allows transferring complexity upper bounds for solving games on static arenas [17].

▶ **Theorem 6.** *Assuming explicit encodings, solving one-player games on temporal graphs is* NL*-complete for reachability conditions and* NP*-complete for generalized reachability. Solving two-player games on temporal graphs is* P*-complete for reachability and* PSPACE*-complete for generalized reachability.*

Note that an explorability game on a temporal graph corresponds to a generalized reachability game on its expansion as outlined above. However, the target sets are no longer singleton sets and therefore the improved (NL. resp. P) upper bounds provided in [17] for (one resp. two-player) generalized reachability with singleton targets do not apply to explorability on temporal graphs. For example, the explorability game on $\mathcal{A}_1$ (Figure 1) corresponds to the generalised reachability game on its expansion (Figure 3) with targets $\{s\} \times \underline{4}$, $\{t\} \times \underline{4}$, $\{u\} \times \underline{4}$, and $\{v\} \times \underline{4}$, where $\underline{4} = \{0, 1, 2, 3, 4\}$ are all possible times up to the horizon.

In contrast to games on static arenas, where explorability is not harder than reachability, we will see that on temporal graphs explorability is as hard as generalized reachability.

▶ **Theorem 7.** *Solving one-player explorability games on an explicit temporal graph is* NP*-complete.*

Proof. The NP-hardness is due to Michail and Spirakis [32, Proposition 2] by reduction from the Hamiltonian Path problem. Indeed, a directed graph with n vertices has a Hamiltonian path if and only if it can be explored in exactly n steps. A matching upper bound can be achieved by stepwise guessing an exploring path of length at most $h(G)$, which is polynomial given that the input graph is explicitly represented. ◀

We now consider the impact of an antagonistic player for explorability on temporal graphs. A key element of our lower bound construction is the interaction of antagonistic choice and the passing of time. Our reduction, from QBF, relies on a time-bounded final "flooding phase" that allows exploration only if sufficient time is left, and which can only be guaranteed by winning the preceding QBF game.

▶ **Theorem 8.** *Solving two-player explorability games on explicit temporal graphs is* PSPACE*-complete.*

Proof. The upper bound for solving explorability games on a temporal arena $\mathcal{A}$ follows from solving the generalized reachability game on the expansion of $\mathcal{A}$, where we have a target set $F_i = \{(v_i, \theta) \mid \forall\ \theta \in \{0, 1, \ldots, h(\mathcal{A})\}\}$ for every vertex v_i of $\mathcal{A}$. Generalized reachability games can be solved in PSPACE [17, Theorem 1].

We provide a matching lower bound using a reduction from QSAT. Given a QBF formula $\Phi = \exists x_1 \forall x_2 \ldots \exists x_n.\ C_1 \wedge C_2 \wedge \ldots \wedge C_k$, we construct a two-player explorability game G_Φ on a temporal graph so that Player 1 wins iff Φ is true. Let φ refer to the matrix $C_1 \wedge C_2 \wedge \ldots \wedge C_k$ of the given formula Φ, that is, each C_i is a disjunction of at most 3 literals. The game G_Φ has vertices $V = V_\forall \uplus V_\exists \uplus \{q_\varphi\} \uplus V_{\text{literals}} \uplus V_{\text{clause}}$, where

- the vertices $V_\exists = \{q_1, q_3, \ldots, q_n\}$, $V_{literals} = \{q_{x_1}, q_{\neg x_1}, q_{x_2}, q_{\neg x_2}, \ldots, q_{x_n}, q_{\neg x_n}\}$ and $V_{clauses} = \{q_{C_1}, q_{C_2}, \ldots, q_{C_k}\}$ are controlled by Player 1.
- the vertices $V_\forall \{q_2, q_4, \ldots, q_{n-1}\}$ and $\{q_\varphi\}$ are controlled by Player 2.

The game is played in 4 phases, each of a fixed length, as follows.

1. In the *initial phase*, from time 0 to $k-1$, the game starts from q_{C_1} and visits all clause vertices of $V_{clauses}$ in order and then ends up in the vertex q_1 corresponding to the first existential quantifier. Formally, the edges available during this phase are, for all $0 \leq i < k$, $E(q_{C_i}, q_{C_{i+1}}) = E(q_{C_k}, q_1) = [0, k)$. Edges of the initial phase are drawn in red in the example in Figure 4.
2. The *selection phase*, from time k to $k + 2n - 1$, the game visits the vertices corresponding to the quantifiers in order of their appearance in the formula. Depending on whether the quantifier is $\exists$ (or $\forall$), Player 1 (respectively Player 2) chooses to visit a vertex corresponding to either a positive or negative literal for each variable. The edges available during this phase are for all $0 \leq i < n$, $E(q_i, q_{x_i}) = E(q_i, q_{\neg x_i}) = E(q_{x_i}, q_{i+1}) = E(q_{\neg x_i}, q_{i+1}) = E(q_{x_n}, q_\varphi) = E(q_{\neg x_n}, q_\varphi) = [k, k + 2n)$. Corresponding edges are in blue in Figure 4 and corresponds to choosing a valuation for the variables in the formula Φ.
3. The *evaluation phase*, at time $k + 2n$ and $k + 2n + 1$, starts from vertex q_φ, where Player 2 selects a clause C_i by moving to vertex q_{C_i}. Formally, we have for all $0 < i \leq k$, $E(q_\varphi, q_{C_i}) = \{k + 2n\}$. From q_{C_i}, Player 1 has the choice to visit a literal vertex q_ℓ at time $k + 2n + 1$ if $\neg\ell$ is contained in the clause C_i, where ℓ is a literal. After this phase, all vertices in $V_\forall \uplus V_\exists \uplus \{q_\varphi\} \uplus V_{\text{clause}}$ have been visited as well as half of $V_{literal}$. In Figure 4, edges for the clause selection and literal selection are in green.
4. The game ends after a *flooding phase* that lasts for exactly $n - 1$ steps starting from time $k + 2n + 2$. That is, all edges become unavailable from time $k + 3n + 1$ onwards. During this phase, an edge is available from a literal vertex q_{ℓ_i} to both literal vertex $q_{x_{i+1}}$ and $q_{\neg x_{i+1}}$. Formally, the edges $(q_{\ell_i}, q_{\ell_{i+1}})$, for all $0 < i < n$ and (q_{ℓ_n}, q_{ℓ_1}), where $\ell_j \in \{x_j, \neg x_j\}$, are available at times $[k + 2n + 2, k + 3n + 1]$. Thus, in this phase Player 1 can visit exactly $n - 1$ of the possibly unexplored vertices. The flooding phase is shown in yellow in Figure 4.

Suppose the QBF formula Φ is true. Then the $\exists$-player has a strategy to assign values to the variables x_i (for odd i) to ensure that $\varphi(\vec{x})$ is true. By following the same choices in the selection phase (moving $q_i \longrightarrow q_{x_i}$ if x_i is set to *true* and $q_i \longrightarrow q_{\neg x_i}$ otherwise), Player 1 guarantees that when q_φ is reached, the jointly chosen variable assignment $\vec{x}$ satisfies φ. At this point the play has already visited exactly one vertex among q_{x_j} and $q_{\neg x_j}$ for all $j \leq n$. Moreover, no matter which vertex q_{C_i} is chosen by Player 2 in the next step, the corresponding clause C_i is also satisfied by assignment $\vec{x}$. By construction, this means there exists a literal $\ell_j \in C_i$ such that vertex q_{ℓ_j} has been visited before. The Player 1 can then move $C_i \longrightarrow q_{\neg\ell_j}$ to end the evaluation phase. Finally, in the flooding phase, Player 1 can freely move to visit the remaining $n - 1$ literal vertices not visited thus far. This guarantees that indeed, all vertices have been visited and Player 1 wins the explorability game.

If the QBF instance Φ is not satisfiable, then the $\forall$-player has a strategy to assign values to the x_{2i} to ensure that $\varphi(\vec{x})$ is false. By following the same choices during the selection phase (moving $q_i \longrightarrow q_{x_i}$ if x_i is set to *true* and $q_i \longrightarrow q_{\neg x_i}$ otherwise), Player 2 guarantees

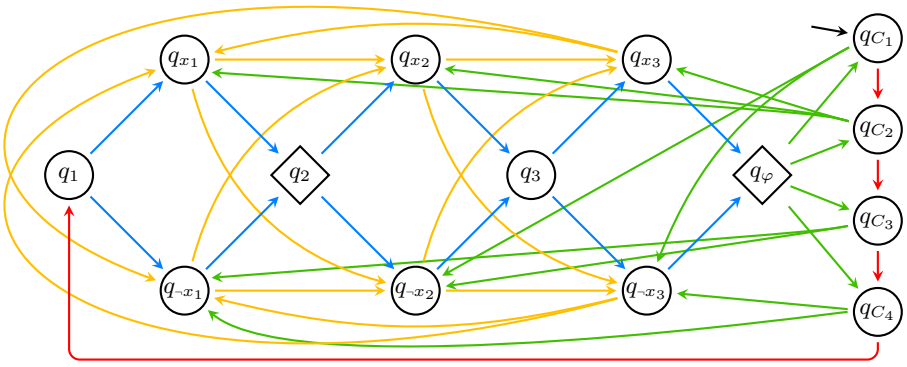


Figure 4 The exploration game for QBF formula $\exists x_1{:}\forall x_2{:}\exists x_3{:}(C_1 = x_2 \vee x_3) \wedge (C_2 = \neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge (C_3 = x_1 \vee x_2) \wedge (C_4 = x_1 \vee x_3)$, where colours encode edge availibility: red=$[0, k)$, blue=$[k, k + 2n)$, green=$[k + 2n, k + 2n + 1]$, yellow=$(k + 2n + 1, k + 3n + 1]$.

that when q_φ is reached, the jointly chosen variable assignment does not satisfy Φ, meaning that there is some clause C_i that is not satisfied. In the evaluation phase, the strategy τ then picks q_{C_i} as the successor from q_φ. Consequently, all successors of q_{C_i} must be vertices that have already been seen in the play. This means that exactly n of the literal vertices are not yet visited. However, the game ends after the following $n - 1$ of the flooding phase. Therefore, one of the literal vertices must be left unexplored. ◀

5 Symbolically represented Temporal Graphs

We now consider games on temporal graphs that are given in a succinct representation. More precisely, we assume that the arena is given as the underlying (static) graph and for every edge, the set of times at which it is available is represented by a formula in the existential fragment of Presburger Arithmetic (∃PA). Recall that evaluating $\varphi(x)$ for a given ∃PA formula φ and time $x \in \mathbb{N}$ is in NP. We call this representation of temporal graphs *symbolic*. Symbolic representations that are equally as powerful as ∃PA, such that they can represent semi-linear sets, include the likes of solving a union of linear equations and commutative context-free grammars [10]. Finding a satisfying valuation for both of these encodings of semi-linear sets has also been shown to be in NP, much like ∃PA. We use ∃PA as the formulae produced are concise and easy to understand after only a brief viewing. There are other concise encodings (such as the inclusion of the universal fragment of PA) that could be used for a symbolic temporal graph, however the problem of finding membership or satisfiability of a given value may be too complex for the encoding to be efficient. For example, satisfiability with respects to the universal fragment of PA is exponential and not in NP.

▶ **Theorem 9.** *Solving one-player symbolic temporal reachability games is* PSPACE*-hard.*

Proof. We reduce QSAT to one-player temporal reachability with symbolic time encoding. Consider a quantified Boolean formula $\Phi = \exists x_1 \forall x_2 \ldots \exists x_n : \varphi$ where φ is a propositional formula with variables in $X = \{x_1, \ldots, x_n\}$. Without loss of generality assume that the quantifiers alternate strictly, starting and ending with existential quantifiers (introduce at most $n + 1$ useless variables otherwise). We show how to construct the symbolic one-player game G_Φ such that Φ is satisfiable if and only if some target vertex of G_Φ can be reached. We use elapse of time to encode valuations of variables of Φ. At a time $\theta \geq 0$, since φ

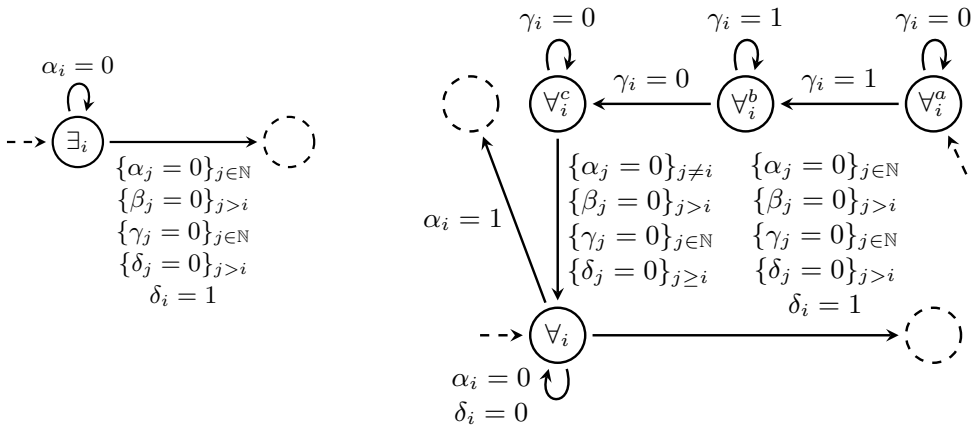


Figure 5 Gadgets to construct G_Φ.

is quantifier-free, determining whether the valuation $\nu_\theta \colon X \to \{0,1\}$ makes Φ true can be checked with a single transition that is available exactly at time θ. The difficulty arises from encoding of the "adversarial" behaviour of universal quantifiers. To do so, we leverage both the graph structure and the Presburger arithmetic.

In our encoding, time is sectored into n segments of 4 bits. The horizon of G_Φ is thus $h(G_\Phi) = 2^{4n}$. For every $i \in \{1,\dots,n\}$ and time $\theta \in \{0,...,2^{4n}-1\}$, we call α_i, β_i, γ_i, and δ_i the four bits of the ith most significant sector, i.e., the bits at indices $(4n-1)-(4i-4)$, $(4n-1)-(4i-3)$, $(4n-1)-(4i-2)$ and $(4n-1)-(4i-1)$ in the binary expansion of θ. In particular, $\alpha_1 = 4n-1$ and $\delta_n = 0$ are respectively the most and the least significant bits. Intuitively, in the sequel we use β_i to represent the chosen value of variable $x_i \in X$, we control overflows of β_i with assumption on α_i and γ_i, and δ_i is an extra bit acting as buffer, to allow spending time without influencing the others. In the sequel, we define constrains on such bits. To do so, for all $k \in \{0,\dots,4n-1\}$ and all $v \in \{0,1\}$, we require the kth least significant bit of a given time $\theta \in \{0,\dots,2^{4n}-1\}$ to be v. This can be expressed thanks to following Presburger formula, which states that the kth least significant bit in the binary expansion of θ equals v.

$$\Psi_{k,v}(\theta) = \exists y_0 : \dots \exists y_{4n} : \exists v_0 : \dots \exists v_{4n-1} : \wedge \begin{cases} \bigwedge_{j=0}^{4n-1}(v_j = 0 \vee v_j = 1) & \text{i.e., } v_j \in \{0,1\} \\ \bigwedge_{j=0}^{4n-1}(y_j = 2y_{j+1} + v_j) & \text{i.e., } y_{j+1} \text{ is } \left\lfloor \frac{\theta}{j+1} \right\rfloor \\ (y_0 = \theta) \wedge (v_k = v) & \text{and } v_k \text{ is } \theta\text{'s } k\text{th bit} \end{cases}$$

In the following, we write $\xi_i = v$, where $\xi \in \{\alpha,\beta,\gamma,\delta\}$ and $v \in \{0,1\}$ as a shorthand for the formula for checking the corresponding bits.

In G_Φ, each quantifier will have a corresponding vertex. If $x_i \in X$ is existentially quantified, the vertex q_i evaluates x_i thanks to a self-loop which also set $\delta_i = 1$. In other words, Player 1 can use the self-looping transition of q_i for spending time (not more than 2^{α_i} steps since α_i must be 0) which in particular leaves him possibility to set the bit β_i as an encoding the value of x_i. Additionally, the non-looping transition of q_i is available only when the bit δ_i is 1. See the left gadget of Figure 5. Observe that, if q_i is entered at time θ where $\delta_i = 0$, there are exactly two times θ_0, θ_1 when q_i can be exited, and θ_0 and θ_1 only differ on the bit β_i. Formally, $\theta_0 = \theta + 2^{(4n-1)-(4i-1)}$, $\theta_1 = \theta_0 + 2^{(4n-1)-(4i-3)}$. This is a direct consequence of the availability of both outgoing transitions of q_i leaving only the bit β_i as degree of freedom to Player 1. We shall prove that vertices encoding existential quantifiers

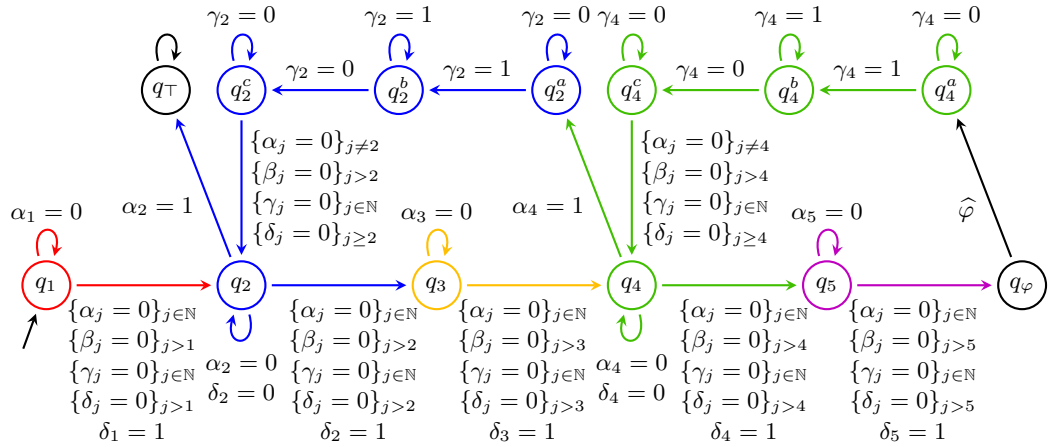


Figure 6 Encoding of a formula with 5 quantifiers to a symbolic one-player game.

are the only source of branching in G_Φ. If $x_i \in X$ is universally quantified, the self-loop of the vertex q_i only serves to set $\delta_i = 1$, the evaluation of x_i being driven by vertices encoding existential quantifiers and auxiliary vertices q_i^a, q_i^b and q_i^c. See the right gadget of Figure 5. The vertex q_i admits two behaviours. If q_i is entered at a time when $\alpha_i = 0$, then q_i exits toward the vertex encoding the next existential quantifier. Otherwise, $\alpha_i = 1$ and then q_i backtracks toward the auxiliary vertices of the previous universal quantifier. The elapsing of time while backtracking will set γ_{i-2} to 1, and then sets it back to 0. If $\beta_{i-2} = 0$, the side effect of backtracking is to switch β_{i-2} from 0 to 1. Otherwise, $\beta_{i-2} = 1$, and the side effect of backtracking is to switch α_{i-2} from 0 to 1, which triggers a further backtrack in q_{i-2}. To end the construction of G_Φ, we add two more vertices q_φ (to check the valuation of Φ encoded by elapse of time) and $q_\top$ (to end the cascade of backtrack). A complete picture is given when Φ has 5 quantifiers in Figure 6. The quantifier-free Presburger formula $\widehat{\varphi}$ is defined as φ where every positive occurrence x_i in φ is replaced by $\beta_i = 1$ in $\widehat{\varphi}$ and every negative occurrence $\neg x_i$ is replaced by $\beta_i = 0$. The size of G_Φ, in particular the encoding of edges availability is discussed at the end of the proof.

Next, we prove that Φ is satisfiable if and only if G_Φ admits a path from q_1 to $q_\top$ composed of exactly 2^{4n} edges, thanks to the unrestricted self-loop on $q_\top$. The argument holds by induction on the odd number $n \geq 1$ of quantifiers. In the base case, $n = 1$ and G_Φ has three vertices q_1, q_φ and $q_\top$. We show that when q_1 is entered at time 0, it can be exited only at time $\theta_0 = 1$ and $\theta_1 = 5$. By construction, q_1 can only be exited toward q_φ that has a single outgoing edge toward $q_\top$. Hence, the edge from q_φ to $q_\top$ can only be traversed at time $\theta_0 + 1 = 2$ or $\theta_1 + 1 = 6$ where $\beta_1 = 0$ and $\beta_1 = 1$ respectively. By definition of $\widehat{\varphi}$, the vertex $q_\top$ is reachable if and only if Φ is satisfiable.

Now, assume that Φ has an odd number $n \geq 3$ of quantifiers. Let $\Phi = \exists x_1 \forall x_2 \Phi'$ where $\Phi' = \exists x_3 \forall x_4 \ldots \exists x_n : \varphi$. For all $v_1, v_2 \in \{\texttt{false}, \texttt{true}\}$, we denote $\Phi'[x_1 \leftarrow v_1, x_2 \leftarrow v_2]$ the formula Φ' where x_1 takes value v_1 and x_2 takes value v_2. Observe that $\Phi'[x_1 \leftarrow v_1, x_2 \leftarrow v_2]$ is a quantified Boolean formula without free-variable. For all $v_1, v_2 \in \{\texttt{false}, \texttt{true}\}$, by induction hypothesis $G_{\Phi'[x_1 \leftarrow v_1, x_2 \leftarrow v_2]}$ admits a path from q_1' to $q_\top'$ composed of exactly $2^{4(n-2)}$ edges if and only if $\Phi'[x_1 \leftarrow v_1, x_2 \leftarrow v_2]$ is satisfiable. Intuitively, we construct such a path of G_Φ from q_1 to $q_\top$ based on a path of $G_{\Phi'}$ from q_1' to $q_\top'$. This is effective since $G_{\Phi'[x_1 \leftarrow v_1, x_2 \leftarrow v_2]}$ is a subgraph of $G_{\Phi[x_1 \leftarrow v_1, x_2 \leftarrow v_2]}$ where q_3 and q_2^a in $G_{\Phi[x_1 \leftarrow v_1, x_2 \leftarrow v_2]}$ match respectively with q_1' and $q_\top'$ in $G_{\Phi'[x_1 \leftarrow v_1, x_2 \leftarrow v_2]}$. In fact, $G_{\Phi'[x_1 \leftarrow v_1, x_2 \leftarrow v_2]}$ has an horizon

of $2^{4(n-2)}$ because Φ' has $n-2$ quantifiers, and q_2^a has a self-loop available at all time in this horizon, that is, $G_{\Phi'[x_1 \leftarrow v_1, x_2 \leftarrow v_2]}$ cannot modify the bits of bit sectors 1 and 2. Also, q_3 is entered only when the $4(n-2)$ least significant bits of the time are zero due to the availability of the edge from q_2 to q_3, and q_2^a is leaved only if the $4(n-2)$ bits overflow due to as a direct consequence of the availability of the edge from q_2^a to q_2^b.

We show that when q_1 is entered at time 0, it can be exited only at time $\theta_0 = 2^{4n-4}$, $\theta_1 = 2^{4n-4} + 2^{4n-2}$. Then q_2 is visited and since in both cases $\alpha_2 = 0$, by construction, the vertex q_3 is visited at time $\theta_0 + 2^{4n-8}$ or $\theta_1 + 2^{4n-8}$ (where δ_2 switched to 1). Let $\theta'_0 = \theta_0 + 2^{4n-8}$ where $\beta_1 = 0$ and $\theta'_1 = \theta_1 + 2^{4n-8}$ where $\beta_1 = 1$. In both cases, $\beta_2 = 0$. For all $k \in \{0,1\}$, by induction hypothesis, there is a path from q_3 at time θ'_k to q_2^a at time $\theta'_k + 2^{4(n-2)} - 1$ if and only if $\Phi'[x_1 \leftarrow k, x_2 \leftarrow 0]$ is satisfiable. If $\Phi'[x_1 \leftarrow 0, x_2 \leftarrow 0]$ and $\Phi'[x_1 \leftarrow 1, x_2 \leftarrow 0]$ are not satisfiable, then G_Φ cannot reach $q_\top$ from q_1 and Φ is not satisfiable. Otherwise, the two paths in G_Φ that reach q_2^a at $\theta'_0 + 2^{4(n-2)} - 1$ and $\theta'_1 + 2^{4(n-2)} - 1$ can be extended to reach q_2 at time $\theta_0 + 2^{4n-6}$ and $\theta_1 + 2^{4n-6}$ respectively (where β_2 switched to 1). Since $\alpha_2 = 0$ in both cases, the vertex q_3 is visited at time $\theta_0 + 2^{4n-6} + 2^{4n-8}$ and $\theta_1 + 2^{4n-6} + 2^{4n-8}$ respectively (where switched to $\delta_2 = 1$). Let $\theta''_0 = \theta_0 + 2^{4n-6} + 2^{4n-8}$ where $\beta_1 = 0$ and $\theta''_1 = \theta_1 + 2^{4n-6} + 2^{4n-8}$ where $\beta_1 = 1$. For all $v_1, v_2 \in \{\texttt{false}, \texttt{true}\}$, by induction hypothesis, there is a path from q_3 at time θ''_k to q_2^a at time $\theta_k + 2^{4(n-2)} - 1$ if and only if $\Phi'[x_1 \leftarrow k, x_2 \leftarrow 1]$ is satisfiable. If $\Phi'[x_1 \leftarrow 0, x_2 \leftarrow 1]$ and $\Phi'[x_1 \leftarrow 1, x_2 \leftarrow 1]$ are not satisfiable, then G_Φ cannot reach $q_\top$ and Φ is not satisfiable. Otherwise, the two paths in G_Φ that reach q_2^a at $\theta''_0 + 2^{4(n-2)} - 1$ and $\theta''_1 + 2^{4(n-2)} - 1$ can be extended to reach q_2 at time $\theta_0 + 2^{4n-5}$ and $\theta_1 + 2^{4n-5}$ respectively (where α_2 switched to 1). Since $\alpha_2 = 1$, the vertex $q_\top$ is visited at time $\theta_0 + 2^{4n-5} + 1$ and $\theta_1 + 2^{4n-5} + 1$ respectively. Hence, G_Φ admit a path from q_1 to $q_\top$ if and only if there is $k \in \{0,1\}$ such that $\Phi'[x_1 \leftarrow k, x_2 \leftarrow 0]$ and $\Phi'[x_1 \leftarrow k, x_2 \leftarrow 1]$ are satisfiable, i.e., if and only if Φ is satisfiable.

It is worth emphasizing that, when G_Φ admits a path from q_1 to $q_\top$, it can be constructed with a memoryless strategy. In the above reduction, for each vertex q_i that encodes an existential quantifier, the inductively constructed strategy aims in q_i at evaluating β_i such that the next time q_φ is visited its outgoing edge will be available. Hence, in q_i, the strategy is solely based on the value $\beta_{i-1}, \ldots, \beta_1$. Since the vertices encoding existential quantifiers are the only source of branching in G_Φ, the strategy is memoryless.

Finally, we show that the size of G_Φ is polynomial in the size of Φ. Here after, the size of the formula corresponds to the number of symbols to write it. The size of a symbolic temporal graphs G, is defined by $|G| = |V| + \sum_{(u,v) \in V^2} |E(u,v)|$. The graph G_Φ has at most $4n+2$ vertices, and all availability are expressible by a Presburger formula of polynomial size in $|\Phi|$. Since the horizon of G_Φ is 2^{4n}, each edge availability can be expressed as a conjunction of at most $4n$ formulas of the form $\Psi_{k,v}$. ◀

▶ **Corollary 10.** *Solving reachability games on one-player symbolic temporal graphs with at most K temporal edges is Σ_m^P-hard, i.e, hard for the m-th level of the polynomial hierarchy, where $K \geq 2\lceil \frac{m}{2} \rceil + 9\lfloor \frac{m}{2} \rfloor + 1$.*

Proof. In the proof of Theorem 9, the number of quantifiers in Φ determines the number of temporal edges used in the constructed symbolic temporal graph. The gadget to encode existential quantifiers uses two temporal edges and the one for universal quantifiers uses nine temporal edges, and there is one edge for the quantifier-free formula φ (see Figure 5). Note that if a quantifier block contains more than m variables, we can blow up the number of bits polynomially by having bits β_i^j for $1 \leq j \leq m$, while keeping the number of temporal edges the same, thus obtaining a reduction from QSAT with a fixed number of quantifiers. ◀

A similar Π_m^P-hardness also holds for a different choice of m. Note that this is in contrast with the PSPACE-hardness proof for two-player reachability games on symbolic temporal graphs, where the problem is PSPACE-hard even with just one temporal edge [17, Theorem 1].

A restriction that makes the problem easier is allowing *waiting*. It is typical in study of temporal graphs to allow waiting for an arbitrary amount of time in temporal paths, i.e., instead of having to take an edge at every timestep, edges are traversed at increasing (but not necessarily consecutive) timesteps. This corresponds to having a loop on all vertices that can be traversed at any time. In case of two-player games, this allows Player 2 to stall infinitely and therefore is not interesting. The following remark states that the explorability on a one-player temporal arena with symbolic encoding becomes easy if waiting is allowed. We show that this holds even for generalized reachability objectives.

▶ **Theorem 11.** *Assuming symbolic encodings, solving one-player games on temporal graphs with waiting is* NP*-complete for reachability, explorability and generalized reachability.*

Proof. The NP lower bound follows from checking satisfiability of an existential Presburger formula, which is known to be NP-complete [37]. Given a ∃PA formula ϕ with no free variables, consider the symbolic temporal graph $V = \{s, t\}$, such that $E(s, t) \stackrel{\text{def}}{=} \phi(x)$ is the only available edge, where x is a variable that does not occur in ϕ. The vertex t is reachable from s if, and only if, ϕ is true. For the upper bound, note that the length of a shortest witnessing path, measured by the number of edges traversed, is at most quadratic in the number of vertices of the graph. The NP upper bound can be shown by guessing the vertices visited along a play, together with the time they are visited first. The intermediate path between two such vertices can only visit linearly many vertices as no vertex is repeated in such a path. If a revisit is necessary, there is another path which waits instead. If a path does exist, then the time at which it is visited will at most be exponentially large and consequentially can be guessed in polynomial time. Therefore, by guessing the path and the times at which corresponding edges are taken, we obtain an NP algorithm. ◀

▶ Remark 12. In [15], Fearnley and Jurdziński introduce counter-stack automata to prove that reachability in two-clock timed automata is PSPACE-complete. The model of counter-stack automata bears similarity with the technique used in proof of Theorem 9. Indeed, both rely on a linear ordering of counters (being bits in our model) and that incrementation of a counter can only be performed when the value of all smaller counters is known. When elapse of time sets a bit to 1, the smaller bits are reset to 0. In [15] counters can be reset, which correspond to reset of clocks. Our proof cannot be derived from [15] because time cannot be reset in our setting. Vice versa, our proof uses existential Presburger formula with arbitrarily many variables which is not known to be expressible with a two-clock automata.

▶ **Theorem 13.** *Solving two-player symbolic temporal generalized reachability games is in* EXP*. It is in* PSPACE *for one-player games.*

Proof. Consider a generalized reachability game G on arena $\mathcal{A} = (V_1, V_2, E)$ and with targets $\mathcal{F} = F_1, \dots F_k$. Note that under symbolic encodings, temporal graphs are ultimately periodic, meaning that there are $b, p \in \mathbb{N}$ so that for every $i > b$, $i \in E(u, v)$ if, and only if, $i + p \in E(u, v)$. This is because the set of times at which any given edge is available is defined using a Presburger formula and thus semilinear [20]. Moreover, the least common multiple of the periods of the individual edges is therefore a period of (all edges in) the temporal graph and we can bound b and p to be at most exponential in the size of the input.

We can therefore construct an at most exponentially larger reachability game G' in which control states record the set of states (of G) that have already been visited, and the time up to and within the period. That is, define the arena $\mathcal{A}' = (V_1', V_2', E')$ where $V' = V \times 2^V \times \{0, 1, \ldots, b+p-1\}$ and with edge relation E' so that $(u, S, \theta) \longrightarrow (v, S \cup \{v\}, \theta')$ if $\theta \in E(u, v)$ and either $\theta' = \theta + 1$ or both $\theta = b + p - 1$ and $\theta' = b$. Player 1 owns a state (u, S, θ) in G' iff she owns u in G. Finally, define the reachability target set in the constructed game G' as

$$F' = \{(v, S, \theta) \in V' : \forall F_j \in \mathcal{F}, \exists v \in F_j, \text{ such that } v \in S\}.$$

Notice that any play from $(v, \emptyset, 0)$ in G' uniquely corresponds to a play from state s at time 0 in the original game G. The second components within control states in G' are non-decreasing and record the set of states visited by the corresponding play in G. By definition of our reachability target F', we see that a play is winning for Player 1 in the constructed reachability game G' if, and only if, the corresponding play is winning for Player 1 in the generalized reachability game G.

Note that the size of G' is $|V| \cdot |2^V| \cdot (b + p)$, where $b, p \leq 2^{\mathcal{O}(|G|)}$. The claim thus follows from the (known) facts that two-, and one-reachability games can be solved in polynomial time and logarithmic space, respectively. ◀

▶ **Corollary 14.** *Assuming symbolic encodings, solving one-player games on temporal graphs is* PSPACE*-complete for reachability, explorability and generalized reachability. Solving two-player games on temporal graphs is* PSPACE*-complete for reachability and* PSPACE*-hard and in* EXP *for explorability and generalized reachability.*

6 Conclusion

We study the complexities of solving explorability games on temporal graphs, splitting into several cases based on the way temporal edges are defined and whether one or two-players are active.

First, we transfer results for games on static graphs and show that there, solving explorability games is NL-complete for the one-player and P-complete for the two-player case.

On explicitly represented temporal graphs in which waiting is permitted, it was known that checking the existence of an exploring path, i.e. the solving one-player explorability games is NP-complete [30]. We show here that even for the more general case where waiting is not always allowed the problem remains in NP. We further show that solving two-player games on explicitly represented temporal graphs is PSPACE-complete for reachability, explorability, and generalized reachability conditions. The existing lower bound for reachability [6] crucially relies on binary encodings and the presence of an antagonistic player.

The main technical contribution of this work is a strong lower bound for symbolically represented temporal graphs: We show that already single-player reachability (and thus explorability) games are PSPACE-hard. Our construction uses specially crafted Presburger constraints to ensure that all possible choices of an "opponent" in a QBF game are verified. Towards upper bounds, we show that even two-player generalized reachability games are solvable in deterministic exponential time, which is better than the exponential *space* procedure one gets for free (by encoding time into the state space) but not quite matching our lower bound. We conjecture that indeed, solving two-player games for each of these three objectives remains in PSPACE, mainly because raising the lower bound remained elusive. To show EXP-hardness one needs to encode non-trivial and recoverable information into timestamps. Our attempts to reduce from CTL Satisfiability or Countdown Games so far

required to either construct Presburger constraints to compare bits that are exponentially far apart in the binary expansion of a given free variable (time), or to construct gadgets that "copy" bitstrings from lower to higher significant bit positions. It is worth noting that any such construction must only be correct for explorability but not for the simpler reachability conditions (which are PSPACE-complete [6]).

On the other hand, showing PSPACE-membership would require either find a polynomially bounded untimed game gadgets to evaluate any given ∃PA formula or to encode the antagonistic behaviours arithmetically.

References

1 Duncan Adamson, Vladimir V. Gusev, Dmitriy Malyshev, and Viktor Zamaraev. Faster Exploration of Some Temporal Graphs. In *1st Symposium on Algorithmic Foundations of Dynamic Networks (SAND 2022)*, volume 221 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5:1–5:10. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.SAND.2022.5.

2 Eleni C. Akrida, George B. Mertzios, Paul G. Spirakis, and Christoforos Raptopoulos. The temporal explorer who returns to the base. *Journal of Computer and System Sciences*, 120:179–193, 2021. doi:10.1016/j.jcss.2021.04.001.

3 Eleni C. Akrida, George B. Mertzios, Paul G. Spirakis, and Viktor Zamaraev. Temporal vertex cover with a sliding time window. *Journal of Computer and System Sciences*, 107:108–123, 2020. doi:10.1016/j.jcss.2019.08.002.

4 Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994. doi:10.1016/0304-3975(94)90010-8.

5 Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 1st edition, 2009.

6 Pete Austin, Sougata Bose, and Patrick Totzke. Parity games on temporal graphs. In *International Conference on Foundations of Software Science and Computational Structures*, pages 79–98. Springer Nature Switzerland, 2024. doi:10.1007/978-3-031-57228-9_5.

7 Benjamin Merlin Bumpus and Kitty Meeks. Edge exploration of temporal graphs. In *Combinatorial Algorithms*, pages 107–121. Springer International Publishing, 2021. doi:10.1007/978-3-030-79987-8_8.

8 Krishnendu Chatterjee, Thomas A. Henzinger, and Vinayak S. Prabhu. Timed Parity Games: Complexity and Robustness. *Logical Methods in Computer Science*, Volume 7, Issue 4, December 2011. doi:10.2168/LMCS-7(4:8)2011.

9 Hongjiang Chen, Pengfei Jiao, Huijun Tang, and Huaming Wu. Temporal graph representation learning with adaptive augmentation contrastive. In *Machine Learning and Knowledge Discovery in Databases: Research Track*, pages 683–699. Springer Nature Switzerland, 2023. doi:10.1007/978-3-031-43415-0_40.

10 Dmitry Chistikov, Christoph Haase, and Simon Halfon. Context-free commutative grammars with integer counters and resets. *Theoretical Computer Science*, 735:147–161, 2018. Reachability Problems 2014: Special Issue. doi:10.1016/j.tcs.2016.06.017.

11 Jean-Lou De Carufel, Paola Flocchini, Nicola Santoro, and Frédéric Simard. Cops & robber on periodic temporal graphs: Characterization and improved bounds. In *Structural Information and Communication Complexity*, pages 386–405. Springer Nature Switzerland, 2023. doi:10.1007/978-3-031-32733-9_17.

12 Argyrios Deligkas, Eduard Eiben, and George Skretas. Minimizing reachability times on temporal graphs via shifting labels. In *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence, IJCAI-23*, pages 5333–5340. International Joint Conferences on Artificial Intelligence Organization, August 2023. doi:10.24963/ijcai.2023/592.

13 Stefan Dobrev, Rastislav Královič, and Euripides Markou. Online graph exploration with advice. In *Structural Information and Communication Complexity*, pages 267–278. Springer Berlin Heidelberg, 2012. doi:10.1007/978-3-642-31104-8_23.

14 Thomas Erlebach, Michael Hoffmann, and Frank Kammer. On temporal graph exploration. *Journal of Computer and System Sciences*, 119:1–18, 2021. doi:10.1016/j.jcss.2021.01.005.

15 John Fearnley and Marcin Jurdziński. Reachability in two-clock timed automata is pspace-complete. In *Automata, Languages, and Programming*, pages 212–223. Springer Berlin Heidelberg, 2013.

16 Nathanaël Fijalkow, Nathalie Bertrand, Patricia Bouyer-Decitre, Romain Brenguier, Arnaud Carayol, John Fearnley, Hugo Gimbert, Florian Horn, Rasmus Ibsen-Jensen, Nicolas Markey, Benjamin Monmege, Petr Novotný, Mickael Randour, Ocan Sankur, Sylvain Schmitz, Olivier Serre, and Mateusz Skomra. Games on graphs, 2023. doi:10.48550/arXiv.2305.10546.

17 Nathanaël Fijalkow and Florian Horn. The surprizing complexity of generalized reachability games, 2012. arXiv:1010.2420.

18 Paola Flocchini, Bernard Mans, and Nicola Santoro. Exploration of periodically varying graphs. In *Algorithms and Computation*, pages 534–543. Springer Berlin Heidelberg, 2009. doi:10.1007/978-3-642-10631-6_55.

19 Janosch Fuchs, Christoph Grüne, and Tom Janßen. The complexity of online graph games. In *SOFSEM 2024: Theory and Practice of Computer Science*, pages 269–282. Springer Nature Switzerland, 2024. doi:10.1007/978-3-031-52113-3_19.

20 Seymour Ginsburg and Edwin H. Spanier. Semigroups, Presburger formulas, and languages. *Pacific Journal of Mathematics*, 16(2):285–296, 1966.

21 Erich Grädel, Wolfgang Thomas, and Thomas Wilke, editors. *Automata Logics, and Infinite Games: A Guide to Current Research*. Springer-Verlag, 2002.

22 Petter Holme and Jari Saramäki. *Temporal Network Theory*. Springer, January 2019. doi:10.1007/978-3-030-23495-9.

23 Marcin Jurdziński and Ashutosh Trivedi. Reachability-time games on timed automata. In *International Colloquium on Automata, Languages and Programming*, pages 838–849. Springer Berlin Heidelberg, 2007.

24 Fabian Kuhn, Nancy Lynch, and Rotem Oshman. Distributed computation in dynamic networks. In *Symposium on Theory of Computing*, STOC '10, pages 513–522. Association for Computing Machinery, 2010. doi:10.1145/1806689.1806760.

25 Oded Maler, Amir Pnueli, and Joseph Sifakis. On the synthesis of discrete controllers for timed systems. In *International Symposium on Theoretical Aspects of Computer Science*, pages 229–242. Springer Berlin Heidelberg, 1995.

26 Nicole Megow, Kurt Mehlhorn, and Pascal Schweitzer. Online graph exploration: New results on old and new algorithms. In *Automata, Languages and Programming*, pages 478–489. Springer Berlin Heidelberg, 2011. doi:10.1007/978-3-642-22012-8_38.

27 George B. Mertzios, Hendrik Molter, Rolf Niedermeier, Viktor Zamaraev, and Philipp Zschoche. Computing maximum matchings in temporal graphs. *Journal of Computer and System Sciences*, 137:1–19, 2023. doi:10.1016/j.jcss.2023.04.005.

28 George B. Mertzios, Hendrik Molter, and Viktor Zamaraev. Sliding window temporal graph coloring. *Journal of Computer and System Sciences*, 120:97–115, 2021. doi:10.1016/j.jcss.2021.03.005.

29 George B. Mertzios, Sotiris Nikoletseas, Christoforos Raptopoulos, and Paul G. Spirakis. Brief Announcement: On the Existence of δ-Temporal Cliques in Random Simple Temporal Graphs. In *3rd Symposium on Algorithmic Foundations of Dynamic Networks (SAND 2024)*, volume 292 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 27:1–27:5. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPIcs.SAND.2024.27.

30 Othon Michail. *An Introduction to Temporal Graphs: An Algorithmic Perspective*, pages 308–343. Springer International Publishing, 2015. doi:10.1007/978-3-319-24024-4_18.

31 Othon Michail, Ioannis Chatzigiannakis, and Paul G Spirakis. Causality, influence, and computation in possibly disconnected synchronous dynamic networks. *Journal of Parallel and Distributed Computing*, 74(1):2016–2026, 2014. doi:10.1016/J.JPDC.2013.07.007.

32 Othon Michail and Paul G. Spirakis. Traveling salesman problems in temporal graphs. *Theoretical Computer Science*, 634:1–23, 2016. doi:10.1016/j.tcs.2016.04.006.

33 Andrzej Pelc. Explorable families of graphs. In *Structural Information and Communication Complexity*, pages 165–177. Springer International Publishing, 2018. doi:10.1007/978-3-030-01325-7_17.

34 A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Annual Symposium on Principles of Programming Languages*, POPL '89, pages 179–190. Association for Computing Machinery, 1989. doi:10.1145/75277.75293.

35 Amir Pnueli. The temporal logic of programs. In *Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 46–57. IEEE Computer Society, 1977. doi:10.1109/SFCS.1977.32.

36 Ramamoorthi Ravi. Rapid rumor ramification: Approximating the minimum broadcast time. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 202–213, 1994.

37 Bruno Scarpellini. Complexity of subcases of presburger arithmetic. *Transactions of the American Mathematical Society*, 284(1):203–218, 1984. doi:10.2307/1999283.

38 Yuejiao Wang, Dajun Daniel Zeng, Qingpeng Zhang, Pengfei Zhao, Xiaoli Wang, Quanyi Wang, Yin Luo, and Zhidong Cao. Adaptively temporal graph convolution model for epidemic prediction of multiple age groups. *Fundamental Research*, 2(2):311–320, 2022. doi:10.1016/j.fmre.2021.07.007.

Model Checking as Program Verification by Abstract Interpretation

Paolo Baldan
Department of Mathematics, University of Padua, Italy

Roberto Bruni
Department of Computer Science, University of Pisa, Italy

Francesco Ranzato
Department of Mathematics, University of Padua, Italy

Diletta Rigo
Department of Mathematics, University of Padua, Italy

Abstract

Abstract interpretation offers a powerful toolset for static analysis, tackling precision, complexity and state-explosion issues. In the literature, state partitioning abstractions based on (bi)simulation and property-preserving state relations have been successfully applied to abstract model checking. Here, we pursue a different track in which model checking is seen as an instance of program verification. To this purpose, we introduce a suitable language – called MOKA (for MOdel checking as abstract interpretation of Kleene Algebras) – which is used to encode temporal formulae as programs. In particular, we show that (universal fragments of) temporal logics, such as ACTL or, more generally, universal μ-calculus can be transformed into MOKA programs. Such programs return all and only the initial states which violate the formula. By applying abstract interpretation to MOKA programs, we pave the way for reusing more general abstractions than partitions as well as for tuning the precision of the abstraction to remove or avoid false alarms. We show how to perform model checking via a program logic that combines under-approximation and abstract interpretation analysis to avoid false alarms. The notion of locally complete abstraction is used to dynamically improve the analysis precision via counterexample-guided domain refinement.

2012 ACM Subject Classification Theory of computation → Modal and temporal logics; Theory of computation → Verification by model checking; Theory of computation → Program verification; Theory of computation → Programming logic; Theory of computation → Abstraction

Keywords and phrases ACTL, μ-calculus, model checking, abstract interpretation, program analysis, local completeness, abstract interpretation repair, domain refinement, Kleene algebra with tests

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2025.8

Related Version *Extended Version*: `https://arxiv.org/abs/2506.05525` [1]

Funding This research was partially funded by the *Italian MUR*, under the PRIN 2022 PNRR project no. P2022HXNSC on "Resource Awareness in Programming: Algebra, Rewriting, and Analysis", and the PNRR project *SEcurity and RIghts In the CyberSpace* (SERICS, PE00000014 – CUP H73C2200089001), by the INdAM-GNCS Projects RISICO (CUP E53C22001930001) and MARQ (CUP E53C24001950001), by a *WhatsApp Research Award* and by an *Amazon Research Award* for *AWS Automated Reasoning*.

1 Introduction

Abstraction is a fundamental craft for mastering complexity. In model checking, abstraction-guided space reduction allows to mitigate the well-known state explosion problem [15]. Abstract interpretation [16, 17] is the de facto standard framework for designing sound analyses. The idea of applying abstract interpretation to model checking has been widely

© Paolo Baldan, Roberto Bruni, Francesco Ranzato, and Diletta Rigo;
licensed under Creative Commons License CC-BY 4.0

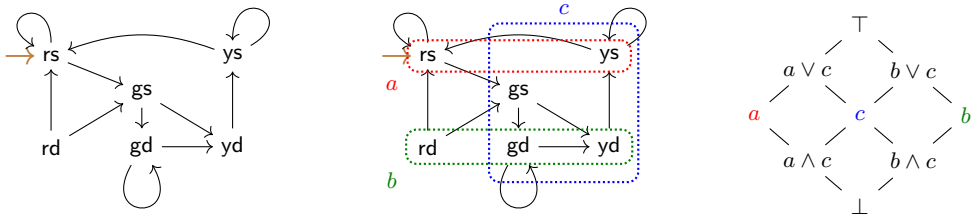
investigated, e.g., [2–5, 10–12, 14, 19–23, 25, 26, 28–30, 33, 35, 37, 38, 40, 44–47, 49, 50, 55] (§ 7 accounts for more closely related work). These approaches often rely on over-approximations that preserve some logical properties, like state partitioning abstractions or simulation-preserving relations.

Objective. In this work, we pursue a different approach, which consists in recasting model checking of temporal formulae directly in terms of program verification, for which the whole tool-suite of abstract interpretation is readily available. To this aim, we exploit an instance of a Kleene algebra with tests and fixpoints (KAF) [36], whose primitives allow to map each temporal formula to a program that can single out all and only the counterexamples to the formula. This program can then be analysed through a sound abstract interpretation that over-approximates the concrete behaviour, so that all possible counterexamples are exposed. However, this over-approximation might not faithfully model the program behaviour in the abstract domain, thus possibly mixing true and false alarms. This lack of precision cannot happen when the abstract interpretation is complete [18]. However, completeness happens quite seldom, and even if, in principle, it can be achieved by refining the abstract domain [27], the most abstract domain refinement ensuring completeness is often way too concrete (it might well coincide with the concrete domain itself).

To remove false alarms, the validity of temporal formulae can be analysed by deriving certain judgements for the corresponding program in a suitable program logic. This enables the use of techniques borrowed from locally complete abstract interpretation [7, 9], where the over-approximation provided by the abstract domain is paired with an under-approximating specification, in the style of O'Hearn's incorrectness logic [43]. This way, any alarm raised by an incorrectness logic proof corresponds to a true counterexample and local completeness guarantees that derivable judgements either exposes some true counterexamples (if any) or proves that there are none. Moreover, the failure of a proof obligation during inference can point out how to dynamically refine the underlying abstraction to enhance the precision and expressiveness of the analysis, a technique called abstract interpretation repair [8].

Methodology (and a toy example). Our main insight is to design a meta-programming language, called MOKA (MOdel checking as abstract interpretation of Kleene Algebras), where suitable temporal formulae can be mapped to. We show how this can be done for ACTL or, more generally, for the single variable μ-calculus without the existential diamond modality. MOKA is a language of regular commands, based on Kleene algebra with tests (KAT) [34] augmented with fixpoint operators (KAF) [36]. MOKA leverages a small set of primitives to extend and filter computation paths. They can be combined with the usual KAF operators of sequential composition, join (i.e., nondeterministic choice), Kleene iteration, and least fixpoint computation. The corresponding programs are then analysed by abstract interpretation.

The language MOKA operates on computation paths $\langle \sigma, \Delta \rangle$, where $\sigma \in \Sigma$ represents the current system state, and $\Delta \in \mathcal{P}(\Sigma)$ represents the set of traversed states. In the following, we informally overload the symbol σ to denote $\langle \sigma, \varnothing \rangle$. Several computation paths can be stacked and unstacked through the MOKA primitives push and pop for dealing with nested temporal formulae. A generic stack is denoted $\langle \sigma, \Delta \rangle :: S$ and we call σ its *current* state. Each temporal formula φ is mapped to a MOKA program $\lfloor \overline{\varphi} \rceil$ that intuitively computes counterexamples to φ, whence the bar over φ in the application of the encoding $\lfloor \cdot \rceil$. Letting $[\![\cdot]\!]$ denote the usual (collecting) denotational semantics, a key result is, therefore, that a state σ satisfies the formula φ iff the execution of $\lfloor \overline{\varphi} \rceil$ on σ returns the empty set. Of course, the results



(a) Behaviour of cars at US traffic light. **(b)** Abstract system. **(c)** Abstract domain.

Figure 1 Variations on the traffic light example from [12].

can be generalised additively to sets of stacks, for which $\lfloor\overline{\varphi}\rceil$ filters out exactly those stacks whose current state satisfies φ. We can thus summarise the semantic correspondence between formulae satisfaction and execution of their MOKA programs by $[\![\lfloor\overline{\varphi}\rceil]\!]P = \{\sigma \in P \mid \sigma \not\models \varphi\}$. As a special case, it follows that $\sigma \models \varphi$ iff $[\![\lfloor\overline{\varphi}\rceil]\!]\{\sigma\} = \varnothing$. For example, for an atomic proposition p, we let $\lfloor\overline{p}\rceil \triangleq \neg\mathsf{p}?$, where $\neg\mathsf{p}?$ is a MOKA primitive that filters out those states where p is valid. As a further example, the MOKA program for the formula $\mathsf{AG}\ \varphi$ (stating that φ along every possible path always holds) is $\lfloor\overline{\mathsf{AG}\ \varphi}\rceil = \mathsf{push}; \mathsf{next}^*; \lfloor\overline{\varphi}\rceil; \mathsf{pop}$.

The intuition is quite simple: next^* generates all the successors by iterating the next operation, and they are filtered by $\lfloor\overline{\varphi}\rceil$, to expose those which fail to satisfy φ. The operations push and pop manage the stack structure. In the example, to attain the semantic correspondence above, whenever a state δ such that $\delta \not\models \varphi$ is reachable from some initial state σ, we want $[\![\lfloor\overline{\mathsf{AG}\ \varphi}\rceil]\!]\{\sigma\}$ to return σ and not δ: the operation push serves to stash σ when looking for δ, and the operation pop to discard δ and unstash σ, to conclude $\sigma \not\models \mathsf{AG}\ \varphi$. To make the next introductory examples easier to follow, we suggest that the reader ignore the push and pop symbols, hence their faded rendering in MOKA programs.

To get a flavour of the proposed approach, we sketch an easy example, which is a variation of [12, Examples 3.4, 3.7]. Consider the transition system in Figure 1a, which models the behaviour of cars at a US traffic light. State names consist of two letters, denoting, respectively, the traffic light status, i.e. red, yellow, green, and the car behaviour, i.e. stop, drive. We model check the validity in the initial state rs of the safety property $\varphi = \mathsf{AG}\ (\neg\mathsf{rd})$, stating that it will never happen that the light is red and the car is driving. This is obviously true for the system, since rd is an unreachable state. We therefore consider the MOKA program $\lfloor\overline{\varphi}\rceil = \mathsf{push}; \mathsf{next}^*; \mathsf{rd}?; \mathsf{pop}$, and compute its semantics $[\![\lfloor\overline{\varphi}\rceil]\!]\{\mathsf{rs}\}$, which turns out to be the empty set, thus allowing us to conclude that $\mathsf{rs} \models \varphi$. In fact, after a few concrete steps of iteration, $[\![\mathsf{rd}?]\!][\![\mathsf{next}^*]\!]\{\mathsf{rs}\} = [\![\mathsf{rd}?]\!]\{\mathsf{rs}, \mathsf{gs}, \mathsf{gd}, \mathsf{yd}, \mathsf{ys}\} = \varnothing$.

How the abstraction works. We show how any sound abstraction of state properties in $\wp(\Sigma)$ of the original system can be lifted to a range of sound abstractions on stacks with varying degrees of precision, in a way that MOKA programs can be analysed by an abstract interpreter, denoted by $[\![\cdot]\!]^\sharp$. The analysis output is always sound, meaning that the set of counterexamples is over-approximated. In particular, letting α denote the abstraction map to an abstract lattice, and $\bot$ the bottom element of the corresponding abstract computational domain, if $[\![\lfloor\overline{\varphi}\rceil]\!]^\sharp\alpha(\{\sigma\}) = \bot$, then $\sigma \models \varphi$ holds. Vice versa, if $[\![\lfloor\overline{\varphi}\rceil]\!]^\sharp\alpha(\{\sigma\}) \neq \bot$, then the abstract analysis might raise a false alarm because, in general, abstractions are not complete.

Back to the previous example, let us consider the (non-partitioning) abstract domain $A \triangleq \{\bot_A, a \wedge c, b \wedge c, a, b, c, a \vee c, b \vee c, \top_A\}$ in Figure 1c, induced by the abstract properties a, b and c, whose concretizations are represented as dotted boxes in Figure 1b. The abstract interpreter computes $[\![\lfloor\overline{\mathsf{AG}\ (\neg\mathsf{rd})}\rceil]\!]^\sharp\alpha(\{\mathsf{rs}\}) = [\![\mathsf{push}; \mathsf{next}^*; \mathsf{rd}?; \mathsf{pop}]\!]^\sharp a = \bot$, thus allowing us

to conclude that $\mathsf{rs} \models \mathsf{AG}\ (\neg\mathsf{rd})$. Since rs has a transition to gs, we have $[\![\mathsf{rd?}]\!]^\sharp [\![\mathsf{next}^*]\!]^\sharp a = [\![\mathsf{rd?}]\!]^\sharp (a \vee c) = \bot$. It is worth remarking that in the abstract domain A, computing $[\![\mathsf{next}^*]\!]^\sharp a$ requires two iterations instead of four, as it happens in the concrete computation. Consider now the formula $\psi = \mathsf{AG}\ (\mathsf{g} \rightarrow \mathsf{AX}\ \mathsf{d})$, namely, "whenever the semaphore is green, then at the next state the car is driving". Spelling out the MOKA encoding, we have: $\lfloor\overline{\psi}\rceil = \mathsf{push}; \mathsf{next}^*; \mathsf{g?}; \mathsf{push}; \mathsf{next}; \neg\mathsf{d?}; \mathsf{pop}; \mathsf{pop}$. Again, ψ holds for the concrete system in Figure 1a, but here the abstract interpretation is imprecise, because $[\![\mathsf{push}; \mathsf{next}^*; \mathsf{g?}]\!]^\sharp \alpha(\{\mathsf{rs}\}) = c::a$ and $[\![\mathsf{push}; \mathsf{next}; \neg\mathsf{d?}]\!]^\sharp (c::a) = (a \vee c)::c::a$, so that $[\![\lfloor\overline{\psi}\rceil]\!]^\sharp \alpha(\{\mathsf{rs}\}) = a \neq \bot$, thus raising a false alarm. As explained above, we exploit push/pop operators to manage the stack structure of domain elements and to recover the (abstract) states from which the computation started when property violations are found. For instance, in the case illustrated above, once the stack $(a \vee c)::c::a$ is obtained, representing an abstract trace where the property fails, its initial abstract state, a, is recovered by applying two successive pop operations.

To eliminate false alarms, we leverage the concept of local completeness in abstract interpretation. Roughly, the idea consists in focusing on the (abstract) computation path produced by some input of interest, and then refining the abstraction only when needed to make it complete *locally* to such computation. More precisely, we apply a variation of local completeness logic (LCL) [7, 9], which is here extended to deal with fixpoint operators. The LCL proof system, parametrised by a generic state abstraction A, works with O'Hearn-like judgements $\vdash_A [P]\ \mathsf{r}\ [Q]$, where r is a program and P, Q denote state properties or, equivalently, the underlying sets of states satisfying those properties. The triple $\vdash_A [P]\ \mathsf{r}\ [Q]$ is valid when Q is an under-approximation of the states reachable by r from P while the abstraction of Q over-approximates such reachable states, and the abstract computation of r on the precondition P is locally complete. Roughly, this can be expressed as $Q \subseteq [\![\mathsf{r}]\!]P \subseteq \gamma \circ \alpha(Q)$. In our setting, where the program $\lfloor\overline{\varphi}\rceil$ associated with a formula φ "returns" all the counterexamples to the validity of φ, an inference of $\vdash_A [P]\ \lfloor\overline{\varphi}\rceil\ [Q]$ shows that $Q \subseteq \{\sigma \in P \mid \sigma \not\models \varphi\} \subseteq \gamma \circ \alpha(Q)$, thus bounding the set of counterexamples to φ in P. Hence, if $Q \neq \varnothing$ then φ does not hold for each $\sigma \in Q \subseteq P$, while φ holds for all states $\sigma \in P \setminus (\gamma \circ \alpha(Q))$. If, instead, $Q = \varnothing$, then $\alpha(Q) = \bot$, and φ holds in the whole P. As LCL derivations cannot succeed with locally incomplete abstractions, if some proof obligation fails, the abstract domain needs to be "fixed" to achieve local completeness. This can be accomplished, e.g., by applying the abstraction repair techniques defined in [8].

For the toy traffic light example, we can derive in LCL the program triple $\vdash_A [\{\mathsf{rs}\}]\ \lfloor\overline{\varphi}\rceil\ [\varnothing]$, that confirms the validity of the formula φ. Vice versa, an attempt to derive the triple $\vdash_A [\{\mathsf{rs}\}]\ \lfloor\overline{\psi}\rceil\ [\varnothing]$ fails because of local incompleteness. Roughly, we can successfully derive $\vdash_A [\{\mathsf{rs}\}]\ \mathsf{next}^*; \mathsf{g?};\ [\{\mathsf{gs}, \mathsf{gd}\}]$, but then the execution of next on $\{\mathsf{gs}, \mathsf{gd}\}$ is not locally complete, because $\alpha([\![\mathsf{next}]\!]\ \{\mathsf{gs}, \mathsf{gd}\}) = \alpha(\{\mathsf{gd}, \mathsf{yd}\}) = b \wedge c$, while $[\![\mathsf{next}]\!]^\sharp \alpha(\{\mathsf{gs}, \mathsf{gd}\}) = [\![\mathsf{next}]\!]^\sharp c = a \vee c$. The abstraction repair procedure of [8] would then lead to refine the abstract domain by adding a new abstract element c_1 to represent the concrete set $\gamma(c_1) = \{\mathsf{gs}, \mathsf{gd}\}$. Since abstract domains must be closed under meets, the addition of c_1 will also entail the addition of $c_1 \wedge b$ such that $\gamma(c_1 \wedge b) = \{\mathsf{gd}\}$. Letting $A_1 \triangleq A \cup \{c_1 \wedge b, c_1\}$, we can still derive $\vdash_{A_1} [\{\mathsf{rs}\}]\ \mathsf{next}^*; \mathsf{g?}\ [\{\mathsf{gs}, \mathsf{gd}\}]$, then $\vdash_{A_1} [\{\mathsf{gs}, \mathsf{gd}\}]\ \mathsf{next}\ [\{\mathsf{gd}, \mathsf{yd}\}]$, and finally $\vdash_{A_1} [\{\mathsf{gd}, \mathsf{yd}\}]\ \neg\mathsf{d?}\ [\varnothing]$, which can be composed together to conclude $\vdash_{A_1} [\{\mathsf{rs}\}]\ \lfloor\overline{\psi}\rceil\ [\varnothing]$. In fact, in A_1 we just have $[\![\lfloor\overline{\psi}\rceil]\!]^\sharp \alpha_1(\{\mathsf{rs}\}) = \bot$.

Original contribution. We set up a theoretical framework for the systematic reduction of model checking of temporal logics to program verification, thereby enabling the re-use of abstract interpretation techniques and verifiers either directly or with minimal effort. This framework consists of:

- MOKA, a meta-programming language in which different temporal logics can be encoded so that the MOKA code for a formula φ computes exactly the counterexamples to φ;
- a systematic technique for lifting abstractions over state properties to abstractions suitable for the static analysis of MOKA programs;
- an extension of the LCL program logic to handle least fixpoints, introducing a form of "false alarm-guided" abstraction refinement loop in the analysis of MOKA programs.

Synopsis. In § 2 we provide some basics about abstract interpretation and the (fragments) of temporal logics considered in the paper. In § 3 we introduce the language MOKA, and then prove in § 4 the key results that relate formula satisfaction with program execution. In § 5 we define a general technique for deriving abstract domains for the static analysis of MOKA programs. In § 6 we showcase how local completeness logic reasoning can be exploited in our framework. In § 7 we discuss related work. Finally, in § 8 we draw some conclusions and sketch future avenues of research. Full proofs of our results and additional material are available in the extended version of this paper [1].

2 Background

A complete lattice is a poset $(L, \leq_L)$ where every subset $X \subseteq L$ has both least upper bound (lub) and greatest lower bound (glb), denoted by $\bigvee_L X$ and $\bigwedge_L X$, respectively, with $\bot_L \triangleq \bigvee_L \varnothing$ and $\top_L \triangleq \bigwedge_L \varnothing$. When no ambiguities can arise, a lattice will be denoted as L and subscripts will be omitted.

Given two complete lattices L_1 and L_2, a function $f : L_1 \to L_2$, is monotone if $x \leq_1 y$ implies $f(x) \leq_2 f(y)$, and additive (resp., co-additive) if it preserves arbitrary lub (resp., glb). Any monotone function $f : L \to L$ on a complete lattice has both a least and a greatest fixpoint, denoted by $\mathrm{lfp}(f)$ and $\mathrm{gfp}(f)$, respectively. The set of functions $f : S \to L$ from a set S to a complete lattice L, denoted L^S, forms a complete lattice when endowed with the pointwise order s.t. $f \leq g$ if for all $s \in S$, $f(s) \leq_L g(s)$. If L_1, L_2 are complete lattices, then $L_1 \times L_2$ is their product lattice endowed with the componentwise order s.t. $(x_1, x_2) \leq (y_1, y_2)$ if $x_1 \leq_1 y_1$ and $x_2 \leq_2 y_2$. We write L^n for the n-ary product of L with itself and $L^+ \triangleq \{\bot, \top\} \cup \bigcup_{n \geq 1} L^n$ for the complete lattice of non-empty finite sequences in L, ordered by $x_1 \dots x_n \leq y_1 \dots y_m$ if $n = m$ and $x_i \leq_L y_i$ for all $i \in [1, n]$, with top $\top$ and bottom $\bot$.

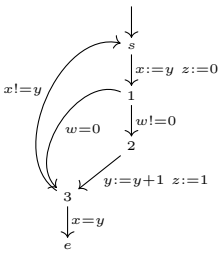
2.1 Abstract Interpretation

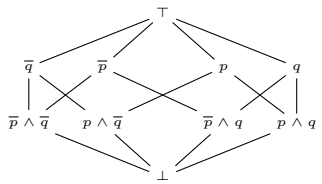
Let us recall the basics of abstract interpretation [17] (see [16] for a thorough account).

Given two complete lattices C and A, called the *concrete* and the *abstract* domain, respectively, a *Galois connection (GC)* $\langle \alpha, \gamma \rangle : C \rightleftarrows A$ is a pair of functions $\alpha : C \to A$ and $\gamma : A \to C$ s.t. $\alpha(c) \leq_A a \Leftrightarrow c \leq_C \gamma(a)$ for any $c \in C$ and $a \in A$.

The function α is referred to as abstraction map and turns out to be additive, while γ is the concretization map which is always co-additive. Intuitively, any abstract element $a \in A$ such that $c \leq \gamma(a)$ is a sound over-approximation for the concrete value c, while the abstraction $\alpha(c)$ is the most precise over-approximation of c in the abstract domain A, i.e., $\alpha(c) = \bigwedge_C \{a \mid c \leq_C \gamma(a)\}$ holds. The notation $A_{\alpha,\gamma}$ denotes an abstract domain endowed with its underlying GC, and we will omit subscripts when α and γ are clear from the context.

```
1  %
2    s  do {
3       x := y; z := 0;  1
4       if (w != 0) then {
5       2   y := y + 1;
6           z := 1; }  3
7       } while (x != y)  e
```





(a) Program c. **(b)** Control flow graph of c. **(c)** Predicate abstraction $\mathbb{P}$.

Figure 2 Program from [4, Figure 1], with control flow graph and predicate abstraction domain.

▶ **Example 2.1** (Image adjunction)**.** Given any function $f : X \to Y$, let $f^{>}$ and $f^{<}$ denote the direct and inverse image of f, respectively. The pair $\langle f^{>}, f^{<}\rangle : \mathcal{P}(X) \to \mathcal{P}(Y)$ is a GC that we refer to as the *image adjunction* (this is an instance of [24, Exercise 7.18]). ⌟

The class of abstract domains on C, denoted by $\mathrm{Abs}(C) \triangleq \{A_{\alpha,\gamma} \mid \langle \alpha, \gamma\rangle : C \rightleftarrows A\}$, can be preordered by the domain refinement relation: $A' \sqsubseteq A$ when $\gamma_A(A) \subseteq \gamma_{A'}(A')$.

▶ **Example 2.2** (Control flow graphs and predicate abstraction)**.** Any program can be represented by its control flow graph (CFG). Let Var be a set of variables valued in $\mathbb{V}$ and denote by $\mathit{Env} = \mathbb{V}^{\mathit{Var}}$ the set of environments. A CFG is a graph (N, E, s, e), where N is a finite set of nodes, representing program points, $s, e \in N$ are the start and end nodes, respectively, and $E \subseteq N \times F \times N$ is a set of edges, labelled over a set of transfer (additive) functions $F \subseteq \mathcal{P}(\mathit{Env})^{\mathcal{P}(\mathit{Env})}$. For example, the program c in Figure 2a is decorated with program points $n \in N = \{s, 1, 2, 3, e\}$, has variables $\mathit{Var} = \{x, y, z, w\}$, and values ranging in the finite domain $\mathbb{V} = \mathbb{Z}_k$ of integers modulo a given $k > 0$. The CFG of c is depicted in Figure 2b.

Predicate abstraction allows to approximate program invariants [32]. Given a set of predicates $\mathit{Pred} \subseteq \mathcal{P}(\mathit{Env})$ (where any $p \in \mathit{Pred}$ has a representation $p = \{\rho \in \mathit{Env} \mid \rho \models p\}$), the predicate abstraction domain $\mathbb{P}$ is defined by adding to Pred the complement predicates $\overline{\mathit{Pred}} \triangleq \{\overline{p} \mid p \in \mathit{Pred}\}$, and then by closing $\mathit{Pred} \cup \overline{\mathit{Pred}}$ under logical conjunction. We define a function $\pi : \mathcal{P}(\mathit{Env}) \to \mathbb{P}$ that associates to each set of environments the strongest predicate it satisfies. For example, given the two predicates $p \triangleq (z = 0)$ and $q \triangleq (x = y)$, the predicate abstraction domain $\mathbb{P}$ induced by the set $\mathit{Pred} = \{p, q\}$ is depicted in Figure 2c. Correspondingly, the product abstraction $\mathbb{P}^N$ allows us to represent the abstract state of the program as a function that associates to each program point n the strongest predicate in $\mathbb{P}$ that holds at n. Hence, for instance, the set of all possible initial states of c, which is $\{(s, xyzw) \mid x, y, z, w \in \mathbb{V}\}$, is represented by the function $(s \mapsto \top, 1 \mapsto \bot, 2 \mapsto \bot, 3 \mapsto \bot, e \mapsto \bot)$, often written $(s \mapsto \top)$, omitting the program points which are mapped to $\bot$. ⌟

An abstract interpreter computes in the underlying abstract domain through correct (and effective) abstract approximations of concrete functions. Given $A_{\alpha,\gamma} \in \mathrm{Abs}(C)$ and a function $f : C \to C$, an abstract function $f^\sharp : A \to A$ is a correct approximation of f if $\alpha \circ f \leq f^\sharp \circ \alpha$, and it is a *complete* approximation of f when $\alpha \circ f = f^\sharp \circ \alpha$. The *best correct approximation* (bca) of f in A, is defined as $f^A \triangleq \alpha \circ f \circ \gamma : A \to A$, and turns out to be the most precise correct abstraction, i.e., $f^A \leq f^\sharp$ for any other correct approximation $f^\sharp$ of f. When $f^\sharp$ is complete, then $f^\sharp = f^A$, thus making completeness an abstract domain property defined by the equation $\alpha \circ f = \alpha \circ f \circ \gamma \circ \alpha$. In program analysis, abstract domains are commonly

endowed with correct but incomplete abstract transfer functions. When completeness holds, the abstract interpreter is as precise as possible for the given abstract domain and cannot raise false alarms when verifying properties that are expressible in the abstract domain [27].

2.2 Transition Systems and Logics

Temporal logics are typically interpreted over unlabeled, finite, directed graphs, whose nodes and edges model states and transitions between them (i.e., state changes), respectively.

A *transition system* T is a tuple $(\Sigma, \mathrm{I}, \mathbf{P}, \rightarrowtriangle, \vdash)$, where Σ is a finite set of states ranged over by σ, $\mathrm{I} \subseteq \Sigma$ is the set of initial states, $\rightarrowtriangle \subseteq \Sigma \times \Sigma$ is the transition relation, $\mathbf{P}$ is a (finite) set of atomic propositions, ranged over by p, and $\vdash \subseteq \Sigma \times \mathbf{P}$ is the satisfaction relation.

We write $\sigma \rightarrowtriangle \sigma'$ instead of $(\sigma, \sigma') \in \rightarrowtriangle$ and let $\mathrm{post}(\sigma) = \{\sigma' \mid \sigma \rightarrowtriangle \sigma'\}$ denote the set of direct successors of $\sigma \in \Sigma$. As common in model checking [13], we consider systems whose transition relation is total, i.e., for all $\sigma \in \Sigma$ there exists $\sigma' \in \Sigma$ such that $\sigma \rightarrowtriangle \sigma'$.

A path is an infinite denumerable state sequence $(\sigma_i)_{i \in \mathbb{N}}$ such that $\sigma_i \rightarrowtriangle \sigma_{i+1}$ for all $i \in \mathbb{N}$.

▶ **Example 2.3** (Control flow graphs as transition systems)**.** A CFG (N, E, s, e) can be viewed as a transition system with states $\Sigma = N \times \mathit{Env}$ and transition relation defined by $(n, \rho) \rightarrowtriangle (n', \rho')$ if there is an edge $(n, f, n') \in E$ such that $\rho' \in f(\{\rho\})$. Additionally, in order to make the transition relation total we add self-loops to all the states (e, ρ) involving the end node e. For instance, by using the shorthand $(n, xyzw)$ for the states in Example 2.2, the transition $(s, 0111) \rightarrowtriangle (1, 1101)$ is induced by the edge $(s, x := y\ z := 0, 1)$ in Figure 2b, while $(3, 1111) \rightarrowtriangle (e, 1111)$ by $(3, x = y, e)$ in Figure 2b.

Let us point out that the predicate abstraction in Example 2.2 naturally lifts to the power-set of states with codomain $A = \mathbb{P}^N$, with the abstraction map $\alpha(X)(n) \triangleq \pi(\{\rho \mid (n, \rho) \in X\})$ for $X \in \mathcal{P}(\Sigma) = \mathcal{P}(N \times \mathit{Env})$. Given $\sigma^\sharp \in A$, we define $\mathit{supp}(\sigma^\sharp) \triangleq \{n \in N \mid \sigma^\sharp(n) \neq \bot\}$. ⌟

ACTL. ACTL is the fragment of CTL whose temporal formulae are universally quantified over all paths leaving the current state. Thus, given a set of atomic propositions $p \in \mathbf{P}$:

$$\mathrm{ACTL} \ni \varphi \ ::= \ p \mid \neg p \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid \mathsf{AX}\ \varphi_1 \mid \mathsf{AF}\ \varphi_1 \mid \mathsf{AG}\ \varphi_1 \mid \varphi_1\ \mathsf{AU}\ \varphi_2$$

▶ **Definition 2.4** (ACTL semantics)**.** *Given a transition system $T = (\Sigma, \mathrm{I}, \mathbf{P}, \rightarrowtriangle, \vdash)$, the semantics $[\![\varphi]\!] \subseteq \Sigma$ of* ACTL *formulae over T is as follows:*

$$\begin{aligned}
[\![p]\!] &\triangleq \{\sigma \in \Sigma \mid \sigma \vdash p\} & [\![\neg p]\!] &\triangleq \{\sigma \in \Sigma \mid \sigma \nvdash p\} \\
[\![\varphi_1 \vee \varphi_2]\!] &\triangleq [\![\varphi_1]\!] \cup [\![\varphi_2]\!] & [\![\varphi_1 \wedge \varphi_2]\!] &\triangleq [\![\varphi_1]\!] \cap [\![\varphi_2]\!] \\
[\![\mathsf{AX}\ \varphi_1]\!] &\triangleq \{\sigma \in \Sigma \mid \forall \sigma'. \sigma \rightarrowtriangle \sigma' \Rightarrow \sigma' \in [\![\varphi_1]\!]\} \\
[\![\mathsf{AF}\ \varphi_1]\!] &\triangleq \{\sigma_0 \in \Sigma \mid \textit{for all path } (\sigma_i)_{i \in \mathbb{N}}\ \exists k \in \mathbb{N}.\ \sigma_k \in [\![\varphi_1]\!]\} \\
[\![\mathsf{AG}\ \varphi_1]\!] &\triangleq \{\sigma_0 \in \Sigma \mid \textit{for all path } (\sigma_i)_{i \in \mathbb{N}}\ \forall j \in \mathbb{N}.\ \sigma_j \in [\![\varphi_1]\!]\} \\
[\![\varphi_1\ \mathsf{AU}\ \varphi_2]\!] &\triangleq \{\sigma_0 \in \Sigma \mid \textit{for all path } (\sigma_i)_{i \in \mathbb{N}}\ \exists k \in \mathbb{N}.\ (\sigma_k \in [\![\varphi_2]\!] \wedge \forall j < k.\ \sigma_j \in [\![\varphi_1]\!])\}
\end{aligned}$$

Universal fragment of single variable μ-calculus. The modal μ-calculus is a well known extension of propositional modal logic with least and greatest fixed point operators. We will focus on its universal fragment only allowing the $\Box$ modal operator that quantifies over all transitions. Moreover, for the sake of simplicity, we restrict to the single variable fragment where, roughly speaking, nested fixpoints cannot have mutual dependencies.

Given a set of atomic propositions $p \in \mathbf{P}$, the $\mu_\Box$-calculus is defined as follows:

$$\mu_\Box \ni \varphi \ ::= \ p \mid \neg p \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid \Box \varphi_1 \mid x \mid \mu x.\varphi_x \mid \nu x.\varphi_x$$

▶ **Definition 2.5** ($\mu_\Box$-calculus semantics). *Given $T = (\Sigma, \mathrm{I}, \mathbf{P}, \rightarrowtriangle, \vdash)$, and a valuation $\mathcal{V} : \mathrm{Var} \to \mathcal{P}(\Sigma)$, the semantics $[\![\varphi]\!]_\mathcal{V} \subseteq \Sigma$ of $\mu_\Box$-calculus formulae over T is as follows:*

$$\begin{aligned}
[\![p]\!]_\mathcal{V} &\triangleq \{\sigma \in \Sigma \mid \sigma \vdash p\} & [\![\neg p]\!]_\mathcal{V} &\triangleq \{\sigma \in \Sigma \mid \sigma \nvdash p\} \\
[\![\varphi_1 \vee \varphi_2]\!]_\mathcal{V} &\triangleq [\![\varphi_1]\!]_\mathcal{V} \cup [\![\varphi_2]\!]_\mathcal{V} & [\![\varphi_1 \wedge \varphi_2]\!]_\mathcal{V} &\triangleq [\![\varphi_1]\!]_\mathcal{V} \cap [\![\varphi_2]\!]_\mathcal{V} \\
[\![\Box\varphi_1]\!]_\mathcal{V} &\triangleq \{\sigma \in \Sigma \mid \forall \sigma'.\sigma \rightarrowtriangle \sigma' \Rightarrow \sigma' \in [\![\varphi_1]\!]_\mathcal{V}\} & [\![x]\!]_\mathcal{V} &\triangleq \mathcal{V}(x) \\
[\![\mu x.\varphi_x]\!]_\mathcal{V} &\triangleq \mathrm{lfp}(\lambda S.[\![\varphi_x]\!]_{\mathcal{V}[x \mapsto S]}) & [\![\nu x.\varphi_x]\!]_\mathcal{V} &\triangleq \mathrm{gfp}(\lambda S.[\![\varphi_x]\!]_{\mathcal{V}[x \mapsto S]})
\end{aligned}$$

where $\mathcal{V}[x \mapsto S]$ is the usual notation for function update.

We write $[\![\varphi]\!]$ instead of $[\![\varphi]\!]_\mathcal{V}$ when the valuation is inessential, and $\sigma \models \varphi$ when $\sigma \in [\![\varphi]\!]$.

3 The Language MOKA

We define a meta-language, called MOKA (for MOdel checking as abstract interpretation of Kleene Algebras), as a (generalised) Kleene Algebra with a set of basic expressions suited for identifying counterexamples to the validity of temporal formulae.

KAF. We rely on Kozen's Kleene Algebra with tests [34] with a Fixpoint operator [36], KAF for short. Given a set Exp of basic expressions e, KAF is defined below:

$$\mathsf{KAF} \ni \mathsf{r} ::= 1 \mid 0 \mid \mathsf{e} \mid \mathsf{r}_1;\mathsf{r}_2 \mid \mathsf{r}_1 \oplus \mathsf{r}_2 \mid \mathsf{r}_1^* \mid \mathsf{X} \mid \mu\mathsf{X}.\mathsf{r}_1$$

The term 1 represents the identity, i.e. no action, 0 represents divergence, $\mathsf{r}_1;\mathsf{r}_2$ represents sequential composition, $\mathsf{r}_1 \oplus \mathsf{r}_2$ represents non-deterministic choice, r_1^* represents the Kleene iteration of r_1, i.e. r_1 performed zero or any finite number of times, X is a variable ranging in a set Var, and $\mu\mathsf{X}.\mathsf{r}_1$ represents the least fixpoint operator with respect to variable X.

Commands are interpreted as functions over a complete lattice C. Given a semantics $(\!|\cdot|\!) : \mathsf{Exp} \to C \to C$ for basic expressions, the semantics of regular expressions $[\![\cdot]\!]_\eta : \mathsf{KAF} \to C \to C$ is inductively defined as follows, where $\eta : \mathsf{Var} \to C \to C$ is an environment:

$$\begin{aligned}
[\![1]\!]_\eta &\triangleq \lambda x.\, x & [\![0]\!]_\eta &\triangleq \lambda x.\, \bot \\
[\![\mathsf{e}]\!]_\eta &\triangleq (\!|\mathsf{e}|\!) & [\![\mathsf{r}_1;\mathsf{r}_2]\!]_\eta &\triangleq [\![\mathsf{r}_2]\!]_\eta \circ [\![\mathsf{r}_1]\!]_\eta \\
[\![\mathsf{r}_1 \oplus \mathsf{r}_2]\!]_\eta &\triangleq [\![\mathsf{r}_1]\!]_\eta \vee [\![\mathsf{r}_2]\!]_\eta & [\![\mathsf{r}_1^*]\!]_\eta &\triangleq \bigvee \left\{ [\![\mathsf{r}_1]\!]_\eta^k \mid k \in \mathbb{N} \right\} \\
[\![\mathsf{X}]\!]_\eta &\triangleq \eta(\mathsf{X}) & [\![\mu\mathsf{X}.\mathsf{r}_1]\!]_\eta &\triangleq \mathrm{lfp}(\lambda f : C \to C.[\![\mathsf{r}_1]\!]_{\eta[\mathsf{X} \mapsto f]})
\end{aligned}$$

It can be seen that the Kleene star r^* can be encoded as a fixpoint $\mu\mathsf{X}.(1 \oplus \mathsf{r};\mathsf{X})$, and, when the semantics of basic expressions $(\!|\cdot|\!)$ is additive, also as $\mathsf{r}^* = \mu\mathsf{X}.(1 \oplus \mathsf{X};\mathsf{r})$. Although redundant, the inclusion of the Kleene star in our syntax is very convenient, as it enables significant simplifications in some encodings, particularly when the full expressiveness of least fixpoint calculations is not necessary (see the remark at the end of Section 4). The same applies to the proof logic studied in Section 6: while basic LCL suffices for the KAF fragment without least fixpoint operator, its extension μLCL is otherwise needed. For closed KAF terms the environment is inessential and we write just $[\![\mathsf{r}]\!]$ instead of $[\![\mathsf{r}]\!]_\eta$.

The Language MOKA. Given a transition system $T = (\Sigma, \mathrm{I}, \mathbf{P}, \rightarrowtriangle, \vdash)$, MOKA is an instance of KAF interpreted over stacks of frames, each frame representing a computation path.

▶ **Definition 3.1** (Frame, stack). *A* frame *is a pair $\langle \sigma, \Delta \rangle \in \Sigma \times \mathcal{P}(\Sigma)$. We denote by F_Σ the set of frames. A* stack *is a finite non-empty sequence of frames, i.e. an element of F_Σ^+, denoted by $\langle \sigma, \Delta \rangle :: S$, where S is a stack or the empty sequence ε.*

A frame $\langle \sigma, \Delta \rangle$ represents a computation in T where σ is the current state and Δ is the set of traversed states, used for loop-detection. The order of the traversed states and their possible multiple occurrences are abstracted away as they are irrelevant when checking the satisfaction of a formula. Frames are stacked to deal with formulae with nested operators.

The language MOKA is defined as instance of KAF with basic expressions for extending and filtering frames, for (de)constructing stacks, and to keep track of fixed-point equations.

▶ **Definition 3.2** (MOKA language). *The language* MOKA *is an instance of* KAF *with the following basic expressions, where* p *ranges over atomic propositions:*

$$\mathsf{e} ::= \mathsf{p?} \mid \neg\mathsf{p?} \mid \mathsf{loop?} \mid \mathsf{next} \mid \mathsf{add} \mid \mathsf{reset} \mid \mathsf{push} \mid \mathsf{pop}$$

The (concrete) semantics of MOKA is given over the powerset of the set of stacks, $C = \mathcal{P}(\mathrm{F}^+_\Sigma)$, ordered by subset inclusion. Although we could focus on stacks of uniform length, we use a larger domain to simplify the notation.

The semantics of MOKA commands $[\![\mathsf{r}]\!]_\eta : \mathcal{P}(\mathrm{F}^+_\Sigma) \to \mathcal{P}(\mathrm{F}^+_\Sigma)$ follows from the general definition for KAF, once we specify the semantics of its basic expressions.

▶ **Definition 3.3** (Basic expression semantics). *Given* $T = (\Sigma, \mathrm{I}, \mathbf{P}, \rightarrowtriangle, \vdash)$*, the semantics of* MOKA *basic expressions is the additive extension of the functions below, where* $\langle \sigma, \Delta \rangle :: S \in \mathrm{F}^+_\Sigma$*:*

$$\begin{array}{ll}
(\!|\mathsf{p?}|\!)\,\{\langle \sigma, \Delta\rangle :: S\} \triangleq \{\langle \sigma, \Delta\rangle :: S \mid \sigma \vdash p\} & (\!|\neg\mathsf{p?}|\!)\,\{\langle \sigma, \Delta\rangle :: S\} \triangleq \{\langle \sigma, \Delta\rangle :: S \mid \sigma \nvdash p\} \\
(\!|\mathsf{loop?}|\!)\,\{\langle \sigma, \Delta\rangle :: S\} \triangleq \{\langle \sigma, \Delta\rangle :: S \mid \sigma \in \Delta\} & (\!|\mathsf{next}|\!)\,\{\langle \sigma, \Delta\rangle :: S\} \triangleq \{\langle \sigma', \Delta\rangle :: S \mid \sigma \rightarrowtriangle \sigma'\} \\
(\!|\mathsf{add}|\!)\,\{\langle \sigma, \Delta\rangle :: S\} \triangleq \{\langle \sigma, \Delta \cup \{\sigma\}\rangle :: S\} & (\!|\mathsf{reset}|\!)\,\{\langle \sigma, \Delta\rangle :: S\} \triangleq \{\langle \sigma, \varnothing\rangle :: S\} \\
(\!|\mathsf{push}|\!)\,\{\langle \sigma, \Delta\rangle :: S\} \triangleq \{\langle \sigma, \Delta\rangle :: \langle \sigma, \Delta\rangle :: S\} & (\!|\mathsf{pop}|\!)\,\{\langle \sigma, \Delta\rangle :: S\} \triangleq \{S \mid S \neq \varepsilon\}
\end{array}$$

The basic expressions p?, ¬p?, loop?, next, add, reset operate on the top frame. The filter p? (resp. ¬p?) checks the validity of the proposition p (resp. $\neg p$), loop? checks if the current state loops back to some state in the trace, next extends the trace by one step in all possible ways, add adds the current state to the trace and reset empties the trace. Instead, push and pop change the stack length: push extends the stack to start a new trace, while pop restores the previous trace from the stack. Given a set of states $X \subseteq \Sigma$, we write $[\![\mathsf{r}]\!]_\eta X$ as a shorthand for $[\![\mathsf{r}]\!]_\eta(\{\langle \sigma, \varnothing\rangle \mid \sigma \in X\})$.

4 Formulae as MOKA programs

We show that ACTL and $\mu_\Box$-calculus formulae φ can be transformed into MOKA programs $\lfloor \overline{\varphi} \rceil$ so that for any transition system T, the semantics of $\lfloor \overline{\varphi} \rceil$ over T consists exactly of the counterexamples to the validity of φ, that is, the states which do not satisfy φ.

▶ **Theorem 4.1** (Model checking as program verification). *Given* $T = (\Sigma, \mathrm{I}, \mathbf{P}, \rightarrowtriangle, \vdash)$*, for any ACTL or* $\mu_\Box$ *formula* φ *and set of stacks* $P \subseteq \mathrm{F}^+_\Sigma$*, it holds* $[\![\lfloor \overline{\varphi} \rceil]\!]_\eta P = \{\langle \sigma, \Delta\rangle :: S \in P \mid \sigma \not\models \varphi\}$.

It follows that $[\![\lfloor \overline{\varphi} \rceil]\!]_\eta P \subseteq P$ and that $[\![\lfloor \overline{\varphi} \rceil]\!]_\eta \{\sigma\} = \varnothing$ iff $\sigma \models \varphi$. It is also worth noting that the semantics of MOKA programs encoding formulae is a lower closure on sets of states, being monotone, reductive and idempotent (and it preserves arbitrary unions) [39, Section 3.2.3]. In this respect, it behaves analogously to the collecting semantics of Boolean tests which filter out memory states, keeping only those that satisfy the condition. As a consequence, similarly to what happens when abstracting Boolean tests, in any abstract domain A approximating sets of states the identity $\lambda a \in A.\, a$, is a correct (over-)approximation of $[\![\lfloor \overline{\varphi} \rceil]\!]_\eta$.

ACTL as MOKA. To each ACTL formula φ we assign the MOKA program $\lfloor\overline{\varphi}\rceil$ as follows:

$$\begin{aligned}
\lfloor\overline{p}\rceil &\triangleq \neg\mathsf{p}? & \lfloor\overline{\neg p}\rceil &\triangleq \mathsf{p}?\\
\lfloor\overline{\varphi_1 \wedge \varphi_2}\rceil &\triangleq \lfloor\overline{\varphi_1}\rceil \oplus \lfloor\overline{\varphi_2}\rceil & \lfloor\overline{\varphi_1 \vee \varphi_2}\rceil &\triangleq \lfloor\overline{\varphi_1}\rceil ; \lfloor\overline{\varphi_2}\rceil\\
\lfloor\overline{\mathsf{AX}\ \varphi}\rceil &\triangleq \mathsf{push}; \mathsf{next}; \lfloor\overline{\varphi}\rceil; \mathsf{pop}\\
\lfloor\overline{\mathsf{AF}\ \varphi}\rceil &\triangleq \lfloor\overline{\varphi}\rceil; \mathsf{push}; \mathsf{reset}; \big(\mathsf{add}; \mathsf{next}; \lfloor\overline{\varphi}\rceil\big)^* ; \mathsf{loop}?; \mathsf{pop}\\
\lfloor\overline{\mathsf{AG}\ \varphi}\rceil &\triangleq \mathsf{push}; \mathsf{next}^*; \lfloor\overline{\varphi}\rceil; \mathsf{pop}\\
\lfloor\overline{\varphi_1\ \mathsf{AU}\ \varphi_2}\rceil &\triangleq \lfloor\overline{\varphi_2}\rceil; \mathsf{push}; \mathsf{reset}; \big(\mathsf{add}; \mathsf{next}; \lfloor\overline{\varphi_2}\rceil\big)^* ; \big(\mathsf{loop}? \oplus \lfloor\overline{\varphi_1}\rceil\big) ; \mathsf{pop}
\end{aligned}$$

The intuition is that MOKA programs $\lfloor\overline{\varphi}\rceil$ act as (negative) filters: applied to a set of frames $\langle\sigma, \Delta\rangle{::}S$, each representing a computation that reached state σ, they filter out states where φ is satisfied, keeping only candidate counterexamples to the validity of φ. In general, when the check requires exploring the future of the current state, a new frame is started with push; if the search is successful, a closing pop recovers the starting state violating the formula. We explain the main clauses defining $\lfloor\overline{\varphi}\rceil$ (see [1, Theorem B.1] for the proof of the correctness of this transform). It turns out that $\langle\sigma, \Delta\rangle{::}S$ is a counterexample for

- AX φ if φ is violated in at least one successor state of σ (as computed by next; $\lfloor\overline{\varphi}\rceil$).
- AF φ if there is an infinite trace where φ never holds. This is encoded by saying that φ does not hold in the current state, i.e. $\lfloor\overline{\varphi}\rceil$, and after that, there is an infinite trace traversing only states (collected through the command $(\mathsf{add}; \mathsf{next}; \lfloor\overline{\varphi}\rceil)^*$) that do not satisfy φ. Since we deal with finite state systems, infinite traces can be identified with looping traces (whence the check loop?); for this to work, after the push we reset the past.
- AG φ if we can reach, by repeatedly applying next, a counterexample to φ.
- φ_1 AU φ_2 if φ_2 does not hold in the current state, i.e. $\lfloor\overline{\varphi_2}\rceil$, and progressing through states that do not satisfy φ_2 with $(\mathsf{add}; \mathsf{next}; \lfloor\overline{\varphi_2}\rceil)^*$, either we detect a maximal (looping) trace or a state where φ_1 does not hold.

▶ **Example 4.2** (Control flow graphs and ACTL encoding)**.** In our running example (see Examples 2.2 and 2.3), we check the property stating that if the program c terminates then the variable z is zero, which in the CFG means *"when the exit node e is reached, $z = 0$"* holds. This is expressed by the ACTL formula $\varphi = \mathsf{AG}\ (n = e \to z = 0)$, where we use $p \to \varphi'$ as syntactic sugar for $\neg p \vee \varphi'$. The corresponding MOKA program is: $\mathsf{m}_\varphi \triangleq \lfloor\overline{\varphi}\rceil = \lfloor\overline{\mathsf{AG}\ (n = e \to z = 0)}\rceil = \mathsf{push}; \mathsf{next}^*; \mathsf{n} = \mathsf{e}?; \mathsf{z} \neq 0?; \mathsf{pop}.$ ⌟

$\mu_\Box$-calculus as MOKA. To each $\mu_\Box$-formula φ we assign the MOKA program $\lfloor\overline{\varphi}\rceil$ as follows:

$$\begin{aligned}
\lfloor\overline{p}\rceil &\triangleq \neg\mathsf{p}? & \lfloor\overline{\neg p}\rceil &\triangleq \mathsf{p}?\\
\lfloor\overline{\varphi_1 \wedge \varphi_2}\rceil &\triangleq \lfloor\overline{\varphi_1}\rceil \oplus \lfloor\overline{\varphi_2}\rceil & \lfloor\overline{\varphi_1 \vee \varphi_2}\rceil &\triangleq \lfloor\overline{\varphi_1}\rceil ; \lfloor\overline{\varphi_2}\rceil\\
\lfloor\overline{\Box\varphi}\rceil &\triangleq \mathsf{push}; \mathsf{next}; \lfloor\overline{\varphi}\rceil; \mathsf{pop} & \lfloor\overline{x}\rceil &\triangleq \mathsf{X}\\
\lfloor\overline{\mu x.\varphi_x}\rceil &\triangleq \mathsf{push}; \mathsf{reset}; \mu\mathsf{X}.\Big(\mathsf{loop}? \oplus (\mathsf{add}; \lfloor\overline{\varphi_x}\rceil)\Big); \mathsf{pop} & \lfloor\overline{\nu x.\varphi_x}\rceil &\triangleq \mu\mathsf{X}.\lfloor\overline{\varphi_x}\rceil
\end{aligned}$$

The second component of a frame $\langle\sigma, \Delta\rangle$ is still used to identify looping computations. This intervenes in the encoding of least fixpoints $\mu x.\varphi_x$: when checking the formula, the current state first logged to the current frame ($\mathsf{add}; \lfloor\overline{\varphi_x}\rceil$ branch); a counterexample is found when we try to verify the fixpoint property in a state where the check has already been tried (filtered by loop?). The encoding of greatest fixpoints $\nu x.\varphi_x$ instead is simpler: searching for counterexamples naturally translates to a least fixpoint, which is offered natively by MOKA.

The correctness of this transform (see [1, Theorem B.2 and Corollary B.3]) leverages a small variation of the tableau construction in [53], with judgements roughly of the form $\sigma, \Delta \vdash \varphi$, meaning that the formula φ holds in a state σ assuming that all the states in Δ

have been visited while checking the current fixpoint subformula. Then, it can be shown that given a formula φ and a stack $\langle\sigma,\Delta\rangle{::}S$, if there is no successful tableau for $\sigma,\Delta\vdash\varphi$ then $[\![\lfloor\overline{\varphi}\rceil]\!]_\eta\,\{\langle\sigma,\Delta\rangle{::}S\}=\{\langle\sigma,\Delta\rangle{::}S\}$ holds, while $[\![\lfloor\overline{\varphi}\rceil]\!]_\eta\,\{\langle\sigma,\Delta\rangle{::}S\}=\varnothing$ otherwise.

▶ **Example 4.3** (Control flow graphs and $\mu_\Box$-calculus encoding)**.** Consider again Examples 2.2 and 2.3. We now check the property *"if after three steps we end up in program point* 3 *then we will loop forever, every* 4 *steps, on this program point 3."* It is known that regular properties of this kind cannot be expressed in ACTL (see, e.g., [54]). This property can be expressed in the $\mu_\Box$-calculus as $\psi\triangleq\Box^3(n=3\ \rightarrow\nu x.\,(n=3\ \wedge\ \Box^4 x))$, where $\Box^k$ is a shortand for $\Box\ldots\Box$ repeated k times. Letting r^k as a shorthand for a k times composition $\mathsf{r};\ldots;\mathsf{r}$ of a MOKA program r, the MOKA program encoding ψ is $\lfloor\overline{\psi}\rceil=(\mathsf{push};\mathsf{next})^3;(\mathsf{n}=3;\lfloor\overline{\nu x.\,(n=3\ \wedge\ \Box^4 x)}\rceil);\mathsf{pop}^3$, with $\lfloor\overline{\nu x.\,(n=3\ \wedge\ \Box^4 x)}\rceil=\mu\mathsf{X}.\,([\mathsf{n}\neq 3?\oplus((\mathsf{push};\mathsf{next})^4;\mathsf{X};\mathsf{pop}^4)])$. ⌟

It is well known that all ACTL formulae can be expressed as $\mu_\Box$-calculus formulae. For example, $\mathsf{AX}\,\varphi=\Box\varphi$, $\mathsf{AF}\,\varphi=\mu x.\,(\varphi\vee\Box x)$, and $\mathsf{AG}\,\varphi=\nu x.\,(\varphi\wedge\Box x)$. Hence one could obtain programs generating counterexamples for ACTL by encoding ACTL in the $\mu_\Box$-calculus and then generating the corresponding program. However, this in general produces programs which are (unnecessarily) more complex than those produced for ACTL formulae. For instance, the program for $\mathsf{AF}\,\varphi$ obtained through the $\mu_\Box$-calculus encoding above would be $\mathsf{push};\mathsf{reset};\mu\mathsf{X}.\Big(\mathsf{loop}?\oplus(\mathsf{add};\lfloor\overline{\varphi}\rceil;\mathsf{push};\mathsf{next};\mathsf{X};\mathsf{pop})\Big);\mathsf{pop}$.

5 Abstract Interpretation of MOKA

We show how to lift an abstraction over the states of a transition system to an abstraction over the domain of stacks. This is achieved stepwise by first considering an abstraction applied to each single stack, and then by merging classes of stacks through a suitable equivalence. The resulting stack abstraction will induce the abstract interpretation of MOKA programs.

Lifting the abstraction. Let $(\Sigma,I,\rightarrowtriangle)$ be a fixed transition system and let $\langle\alpha,\gamma\rangle:\mathcal{P}(\Sigma)\rightleftarrows A$ be an abstraction of state properties. We consider the lattice $\mathrm{F}_A\triangleq A\times A$, with componentwise order, whose elements $\langle\sigma^\sharp,\delta^\sharp\rangle$ are called *abstract frames*, and, in turn, F_A^+ whose elements $\langle\sigma^\sharp,\delta^\sharp\rangle{::}S^\sharp$ are called *abstract stacks*. As in the concrete case, we abbreviate $\langle\sigma^\sharp,\bot_A\rangle$ as $\sigma^\sharp$.

The abstraction map α on state properties can be extended to a frame abstraction, that by abusing the notation, we still denote by α, and is defined by $\alpha(\langle\sigma,\Delta\rangle)\triangleq\langle\alpha(\{\sigma\}),\alpha(\Delta)\rangle$. In turn, the abstraction is inductively defined on stacks as $\alpha(\langle\sigma,\Delta\rangle{::}S^n)\triangleq\alpha(\langle\sigma,\Delta\rangle){::}\alpha(S^n)$.

Now, a set of stacks is abstracted to a set of abstract stacks by first applying α pointwise using the image adjunction (see Example 2.1) and then joining classes of abstract stacks in a way which is parameterised by a suitable equivalence. Given an equivalence $\sim\,\subseteq L\times L$, we let $[x]_\sim$ denote the equivalence class of $x\in L$ w.r.t. $\sim$. Given $x\in L$ we let $\downarrow x\triangleq\{y\in L\mid y\leq x\}$.

▶ **Definition 5.1** (Equivalence adjunction)**.** *Given a complete lattice L, a* compatible equivalence *is an equivalence $\sim\subseteq L\times L$ such that for all $x\in L$, it holds that $[x]_\sim$ is closed by joins of non-empty subsets. Let $\mathcal{P}(L)_\sim\triangleq\{X\in\mathcal{P}(L)\mid\forall x\in X.\,[x]_\sim\cap X=\{x\}\}$, ordered as follows: $X\leq_\sim Y$ if for all $x\in X$ there is $y\in Y$ such that $x\sim y$ and $x\leq y$.*
Then, the pair $\langle\alpha_\sim,\gamma_\sim\rangle:\mathcal{P}(L)\rightleftarrows\mathcal{P}(L)_\sim$ defined, for $X\in\mathcal{P}(L)$, $Y\in\mathcal{P}(L)_\sim$, by

$$\alpha_\sim(X)\triangleq\left\{\bigvee(X\cap[x]_\sim)\mid x\in X\right\}\qquad\gamma_\sim(Y)\triangleq\bigcup\{[y]_\sim\cap\downarrow y\mid y\in Y\}$$

is a Galois connection, called equivalence adjunction.

In words, $\mathcal{P}(L)_\sim \subseteq \mathcal{P}(L)$ consists of the subsets of L containing at most one representative for each equivalence class, and $\alpha_\sim(X)$ abstracts X in the subset (in $\mathcal{P}(L)_\sim$) consisting of the least upper bounds of $\sim$-equivalent elements in X.

We can now define the stack abstraction parameterised by an equivalence on the lattice of abstract frames. Given an equivalence $\sim \subseteq X \times X$ on any set X, $\sim_+$ denotes the equivalence on X^+ induced by $\sim$ as follows: for all $x_1 \cdots x_n, y_1 \cdots y_m \in A^+$, $x_1 \dots x_n \sim_+ y_1 \dots y_m \in X^+$ if $n = m$ and $x_i \sim y_i$ for all $i \in \{1, \dots, n\}$.

▶ **Definition 5.2** (Stack abstraction). *Let $\sim$ be a compatible equivalence on the abstract domain A and let us extend it to the lattice of abstract frames F_A by $\langle \sigma_1^\sharp, \delta_1^\sharp \rangle \sim \langle \sigma_2^\sharp, \delta_2^\sharp \rangle$ if $\sigma_1^\sharp \sim \sigma_2^\sharp$. The* $\sim$-stack abstraction *is the domain $A^s_\sim \triangleq \mathcal{P}(\mathrm{F}_A^+)_\sim$ along with the Galois connection $\langle \alpha^s_\sim, \gamma^s_\sim \rangle : \mathcal{P}(\mathrm{F}_\Sigma^+) \rightleftarrows A^s_\sim$ defined by $\alpha^s_\sim \triangleq \alpha_{\sim_+} \circ \alpha^>$ and $\gamma^s_\sim \triangleq \alpha^< \circ \gamma_{\sim_+}$.*

In words, given a set X of stacks, its abstraction $\alpha^s_\sim(X)$ first applies pointwise the underlying α to each stack in X, and then joins equivalent abstract stacks. As corner cases we can have the identity relation $\sim = \mathrm{id}$, which joins abstract frames with identical first component, and $\sim = A \times A$, the trivial relation, which joins all abstract stacks into one.

An abstract interpreter for counterexamples. Given an abstraction for state properties $\langle \alpha, \gamma \rangle : \mathcal{P}(\Sigma) \rightleftarrows A$ and a compatible equivalence on A, following the standard approach in abstract interpretation [16], we consider an abstract semantics $[\![\mathsf{r}]\!]^\sharp_\eta : A^s_\sim \to A^s_\sim$, defined inductively as explained in § 3 and using as abstract semantics for basic expressions e their BCAs on $A^s_\sim$, denoted by $(\!|\mathsf{e}|\!)^{A^s_\sim}$. We write $[\![\mathsf{r}]\!]^\sharp$ instead of $[\![\mathsf{r}]\!]^\sharp_\eta$ when the abstract environment is inessential. Notably, the BCAs of basic expressions independent from the underlying system can be effectively defined, and some of them result to be complete.

▶ **Theorem 5.3** (Basic abstract operations). *Let $\sim$ be a compatible equivalence on the abstract domain A. The BCAs of the basic expressions* add, reset, push, pop *for the $\sim$-stack abstraction are as follows: for all $\langle \sigma^\sharp, \delta^\sharp \rangle :: S^\sharp \in \mathrm{F}_A^+$*

$$(\!|\mathsf{add}|\!)^{A^s_\sim}\left\{\langle \sigma^\sharp, \delta^\sharp\rangle :: S^\sharp\right\} = \left\{\langle \sigma^\sharp, \delta^\sharp \vee \sigma^\sharp\rangle :: S^\sharp\right\} \qquad (\!|\mathsf{reset}|\!)^{A^s_\sim}\left\{\langle \sigma^\sharp, \delta^\sharp\rangle :: S^\sharp\right\} = \left\{\langle \sigma^\sharp, \bot\rangle :: S^\sharp\right\}$$
$$(\!|\mathsf{push}|\!)^{A^s_\sim}\left\{\langle \sigma^\sharp, \delta^\sharp\rangle :: S^\sharp\right\} = \left\{\langle \sigma^\sharp, \delta^\sharp\rangle :: \langle \sigma^\sharp, \delta^\sharp\rangle :: S^\sharp\right\} \qquad (\!|\mathsf{pop}|\!)^{A^s_\sim}\left\{\langle \sigma^\sharp, \delta^\sharp\rangle :: S^\sharp\right\} = \left\{S^\sharp\right\}$$

and the operations reset, push, pop *are globally complete. Moreover*

$$(\!|\mathsf{p?}|\!)^{A^s_\sim}\left\{\langle \sigma^\sharp, \delta^\sharp\rangle :: S^\sharp\right\} \leq_\sim \left\{\langle \sigma^\sharp \wedge \mathsf{p?}^{A^s_\sim}, \delta^\sharp\rangle :: S^\sharp\right\}$$
$$(\!|\neg\mathsf{p?}|\!)^{A^s_\sim}\left\{\langle \sigma^\sharp, \delta^\sharp\rangle :: S^\sharp\right\} \leq_\sim \left\{\langle \sigma^\sharp \wedge \neg\mathsf{p?}^{A^s_\sim}, \delta^\sharp\rangle :: S^\sharp\right\}$$
$$(\!|\mathsf{loop?}|\!)^{A^s_\sim}\left\{\langle \sigma^\sharp, \delta^\sharp\rangle :: S^\sharp\right\} \leq_\sim \left\{\langle \sigma^\sharp, \delta^\sharp\rangle :: S^\sharp \mid \sigma^\sharp \wedge \delta^\sharp \neq \bot\right\}$$

where $\mathsf{p?}^{A^s_\sim} = \alpha([\![\mathsf{p?}]\!]\Sigma)$ is the abstraction of the set of concrete states satisfying p (and similarly for $\neg\mathsf{p?}^{A^s_\sim}$).

For loop? one can get $(\!|\mathsf{loop?}|\!)^{A^s_\sim}\left\{\langle \sigma^\sharp, \delta^\sharp\rangle :: S^\sharp\right\} = \left\{\langle \sigma^\sharp \wedge \delta^\sharp, \delta^\sharp\rangle :: S^\sharp \mid \sigma^\sharp \wedge \delta^\sharp \neq \bot\right\}$ under additional conditions on $\sim$ (cf. [1, Definition C.6]).

The soundness-by-design of this abstract semantics entails a sound program verification.

▶ **Proposition 5.4** (Program verification for satisfaction). *Given a transition system $T = (\Sigma, \mathrm{I}, \mathbf{P}, \rightarrowtriangle, \vdash)$, for all ACTL or $\mu_\Box$-calculus formulae φ, abstract domain A, and a compatible equivalence $\sim$ on A, if $[\![\lfloor\overline{\varphi}\rceil]\!]^\sharp_\eta(\alpha^s_\sim(I)) = \bot$ then, for all $\sigma \in I$, we have $\sigma \models \varphi$.*

As in the concrete case, the abstract semantics of programs encoding formulae is a lower closure. This, together with the observation that the semantics only depends on the top frame of a stack, is relevant for the concrete implementation.

▶ Remark 5.5. We recalled in § 4 that for any Boolean test b and abstract domain A, the identity function $\lambda a \in A.\, a$ is a correct approximation of the filtering semantics $[\![b]\!]$ of b. This function can be enhanced to $\lambda a \in A.\, \alpha([\![b]\!]\Sigma) \wedge_A a$, which is also a correct approximation of $[\![b]\!]$ [39, Section 4.1]. If A is a partitioning abstraction then one can show that $\lambda a \in A.\, \alpha([\![b]\!]\Sigma) \wedge_A a$ is indeed the *best* correct approximation of $[\![b]\!]$. However, this property does not hold in general, even assuming that A is a disjunctive abstraction, i.e., the additivity of γ (this is substantiated in [1, Example C.10]). ⌟

▶ **Example 5.6** (Control flow graphs and abstract frames)**.** In our running example, abstract frames are of the shape $A \times A = \mathbb{P}^N \times \mathbb{P}^N$. The abstraction of singletons is $\alpha(\{(n, \rho)\}) = (n \mapsto a)$ with $a(n') = \bot$ for all $n' \neq n$. With the aim of joining past states when they correspond to the same program point, we consider the equivalence $\sim \subseteq \mathbb{P}^N \times \mathbb{P}^N$ on abstract frames defined by $\sigma_1^\sharp \sim \sigma_2^\sharp$ when $supp(\sigma_1^\sharp) = supp(\sigma_2^\sharp)$. The property $\psi = \Box^3(n = 3 \ \rightarrow \nu x.\, (n = 3 \ \wedge \ \Box^4 x))$ from Example 4.3 holds in the system, and we can prove it with this abstraction. Let $\sigma^\sharp = (s \mapsto \top) = \alpha(\{(s, xyzw)\})$ we can compute $[\![\lfloor \overline{\psi} \rceil]\!]^\sharp_\eta(\sigma^\sharp) = \bot$, implying that the formula holds from any initial states (convergence is after a single full iteration). Instead, the abstract computation for $\varphi = \mathsf{AG}\ (n = e \rightarrow z = 0)$ from Example 4.2 yields a false positive. ⌟

6 Locally Complete Analyses

If the abstract interpretation of a MOKA program returns an alarm, i.e., $[\![\lfloor \overline{\psi} \rceil]\!]^\sharp \alpha^s_\sim(I) \neq \bot$, then any initial state in the concretisation of $[\![\lfloor \overline{\psi} \rceil]\!]^\sharp \alpha^s_\sim(I)$ is a candidate counterexample to the validity of ψ. However, due to over-approximation, we cannot distinguish spurious counterexamples from true ones. Here, we discuss how to combine under- and over-approximation for the analysis of MOKA programs $\lfloor \overline{\psi} \rceil$ to overcome this problem. In particular we leverage Local Completeness Logic (LCL) [7, 9] possibly paired with Abstract Interpretation Repair (AIR) strategies [8], to improve the analysis precision.

LCL. Most abstract domains are not globally complete for program analysis, so that the corresponding analyses may well yield false alarms. Accordingly, [7, 9] studies how completeness can be locally weakened to an analysis of interest: given a function $f : C \to C$, an abstract domain $A_{\alpha,\gamma} \in \mathrm{Abs}(C)$ is *locally complete on a value* $c \in C$, denoted by $\mathbb{C}^A_c(f)$, when $\alpha \circ f(c) = \alpha \circ f \circ \gamma \circ \alpha(c)$ (hence global completeness amounts to $\mathbb{C}^A_c(f)$ for all $c \in C$). Intuitively, the absence of false alarms in an abstract computation comes as a consequence of the local completeness of the abstract transfer functions on the traversed concrete states.

Moreover, local completeness is a convex property, and this allows to check local completeness on suitable under-approximations $u \leq c$ such that $\alpha(u) = \alpha(c)$, as $\mathbb{C}^A_u(f)$ implies $\mathbb{C}^A_c(f)$. The work [7, 9] implements this idea through LCL, an under-approximating program logic (in the style of incorrectness logic [43]), parameterised by the abstract domain A which provides an over-approximation. The LCL proof rules for KAT programs are recalled in Table 1. A provable LCL triple $\vdash_A [P]\ \mathsf{r}\ [Q]$ ensures that each state satisfying Q is reachable from some state satisfying P, and also guarantees that Q and $[\![\mathsf{r}]\!]P$ have the same abstraction in A. This means that the LCL program logic is sound w.r.t. the following notion of validity.

▶ **Definition 6.1** (Valid LCL triples)**.** *Let $P, Q \in C$ and* r *be a* KAT *program. A program triple $\vdash_A [P]\ \mathsf{r}\ [Q]$ is valid if $\mathbb{C}^A_P(\mathsf{r}) \ \wedge \ Q \leq [\![\mathsf{r}]\!]P \ \wedge \ \alpha([\![\mathsf{r}]\!]P) = \alpha(Q)$.*

The condition $\mathbb{C}^A_P(\mathsf{r})$ states that A is locally complete for $[\![\mathsf{r}]\!]$ on P, while $Q \leq [\![\mathsf{r}]\!]P$ that Q under-approximates $[\![\mathsf{r}]\!]P$ and $\alpha([\![\mathsf{r}]\!]P) = \alpha(Q)$ that $\gamma(\alpha(Q))$ over-approximates $[\![\mathsf{r}]\!]P$.

Table 1 Rules of the Local Completeness Logic [9].

$$\frac{\mathbb{C}^A_P(\mathsf{e})}{\vdash_A [P]\ \mathsf{e}\ [[\![\mathsf{e}]\!]P]}\ \text{(transfer)} \qquad \frac{P' \le P \le \gamma(\alpha(P')) \quad Q \le Q' \le \gamma(\alpha(Q)) \quad \vdash_A [P']\ \mathsf{r}\ [Q']}{\vdash_A [P]\ \mathsf{r}\ [Q]}\ \text{(relax)}$$

$$\frac{\vdash_A [P]\ \mathsf{r}_1\ [W] \quad \vdash_A [W]\ \mathsf{r}_2\ [Q]}{\vdash_A [P]\ \mathsf{r}_1;\mathsf{r}_2\ [Q]}\ \text{(seq)} \qquad \frac{\vdash_A [P]\ \mathsf{r}_1\ [Q_1] \quad \vdash_A [P]\ \mathsf{r}_2\ [Q_2]}{\vdash_A [P]\ \mathsf{r}_1 \oplus \mathsf{r}_2\ [Q_1 \vee Q_2]}\ \text{(join)}$$

$$\frac{\vdash_A [P]\ \mathsf{r}\ [W] \quad \vdash_A [P \vee W]\ \mathsf{r}^*\ [Q]}{\vdash_A [P]\ \mathsf{r}^*\ [Q]}\ \text{(rec)} \qquad \frac{\vdash_A [P]\ \mathsf{r}\ [Q] \quad Q \le \gamma(\alpha(P))}{\vdash_A [P]\ \mathsf{r}^*\ [P \vee Q]}\ \text{(iterate)}$$

Table 2 Novel rules for μLCL.

$$\frac{}{\vdash_A [P]\ \mu^0\mathsf{X}.\mathsf{r}\ [\bot]}\ (\mu^0) \qquad \frac{\vdash_A [P]\ \mathsf{r}[\mu^n\mathsf{X}.\mathsf{r}/\mathsf{X}]\ [Q]}{\vdash_A [P]\ \mu^{n+1}\mathsf{X}.\mathsf{r}\ [Q]}\ (\mu^+) \qquad \frac{[\![\mu\mathsf{X}.\mathsf{r}]\!]^\sharp\alpha(P) \le_A \alpha(Q) \quad \vdash_A [P]\ \mu^n\mathsf{X}.\mathsf{r}\ [Q]}{\vdash_A [P]\ \mu\mathsf{X}.\mathsf{r}\ [Q]}\ \text{(fix)}$$

LCL for MOKA. The LCL proof system has been defined for KAT programs only, hence it is enough for MOKA programs induced by ACTL formulae. For analysing general KAF and thus MOKA programs, we extend it to μLCL including the inference rules in Table 2, where $\mathsf{r}[\mathsf{s}/\mathsf{X}]$ denotes the capture-avoiding substitution of the free occurrences of X for s in r and the term $\mu^n\mathsf{X}.\mathsf{r}$ represents the n-th fixpoint approximant with the expected semantics ($[\![\mu^0\mathsf{X}.\mathsf{r}]\!]_\eta \triangleq \lambda x.\bot$ and $[\![\mu^{n+1}\mathsf{X}.\mathsf{r}]\!]_\eta \triangleq [\![\mathsf{r}]\!]_{\eta[\mathsf{X}\mapsto[\![\mu^n\mathsf{X}.\mathsf{r}]\!]_\eta]}$). The syntax $\mu^n\mathsf{X}.\mathsf{r}$ is used only in μLCL derivations. Intuitively, since $\mu^0\mathsf{X}.\mathsf{r}$ behaves as $\mathsf{0}$, the unique valid judgement is $\vdash_A [P]\ \mu^0\mathsf{X}.\mathsf{r}\ [\bot]$. The rule (μ^+) serves just to unfold $\mu^{n+1}\mathsf{X}.\mathsf{r}$ as many times as needed. The key rule is (fix) which can be used whenever the n-th approximant provides enough information: in case the premise holds, by local completeness we will have $[\![\mu^n\mathsf{X}.\mathsf{r}]\!]^\sharp\alpha(P) = \alpha(Q)$. Since it is always the case that $[\![\mu^n\mathsf{X}.\mathsf{r}]\!]^\sharp\alpha(P) \le_A [\![\mu\mathsf{X}.\mathsf{r}]\!]^\sharp\alpha(P)$, if the side condition $[\![\mu\mathsf{X}.\mathsf{r}]\!]^\sharp\alpha(P) \le_A \alpha(Q)$ holds, then local completeness for the fixpoint term $\mu\mathsf{X}.\mathsf{r}$ can be inferred. These novel rules are sound, in the sense of preserving validity (Definition 6.1).

▶ Remark 6.2. When a least fixpoint appears in $\lfloor\overline{\varphi}\rceil$, we can exploit the following heuristic for choosing the value of n: if the abstract domain satisfies the Ascending Chain Condition (ACC) then, for every $P \in C$ there is $n_P \in \mathbb{N}$ such that $[\![\mu^{n_P}\mathsf{X}.\mathsf{r}]\!]^\sharp\alpha(P) = [\![\mu\mathsf{X}.\mathsf{r}]\!]^\sharp\alpha(P)$. Then, by taking any $n \ge n_P$ such that $\vdash_A [P]\ \mu^n\mathsf{X}.\mathsf{r}\ [Q]$ is provable, the condition $[\![\mu\mathsf{X}.\mathsf{r}]\!]^\sharp\alpha(P) \le_A \alpha(Q)$ is readily satisfied, so that $\vdash_A [P]\ \mu\mathsf{X}.\mathsf{r}\ [Q]$ is valid. More precisely, the rule below is sound:

$$\frac{\vdash_A [P]\ \mu^n\mathsf{X}.\mathsf{r}\ [Q] \qquad [\![\mu^n\mathsf{X}.\mathsf{r}]\!]^\sharp\alpha(P) = [\![\mu\mathsf{X}.\mathsf{r}]\!]^\sharp\alpha(P)}{\vdash_A [P]\ \mu\mathsf{X}.\mathsf{r}\ [Q]}\ \text{(afix)}$$ ⌟

The following result relating μLCL proofs with validity of formulae follows as an easy consequence of [9, Corollary 5.6] and Theorem 4.1.

▶ **Corollary 6.3** (Precision). *Let $T = (\Sigma, \mathrm{I}, \mathbf{P}, \rightarrowtriangle, \vdash)$ be a transition system. For all ACTL or $\mu_\Box$-calculus formulae φ, abstract domain A and compatible equivalence $\sim$ on A, if the triple $\vdash_{A^s_\sim} [I]\ \lfloor\overline{\varphi}\rceil\ [Q]$ is derivable, then $Q \subseteq I$ and*

- *$Q = \varnothing$ if and only if for all $\sigma \in I$ we have $\sigma \models \varphi$;*
- *if $Q \neq \varnothing$ then for all $\sigma \in Q$ we have $\sigma \not\models \varphi$.*

▶ **Example 6.4** (Control flow graphs and LCL derivations)**.** As pointed out in Example 5.6 the computation of the program m_φ, which encodes in MOKA the property $\varphi = \mathsf{AG}\ (n = e \to z = 0)$ (see Example 4.2), yields a false alarm when we use the abstract domain A depicted in Figure 2c. Consider as set of initial states $I_0 = \{\langle(s, 0100), \varnothing\rangle, \langle(s, 0011), \varnothing\rangle\}$, whose abstraction covers the set of possible initial states $\{\langle(s, xyzw), \varnothing\rangle \mid x, y, z, w \in \mathbb{Z}_k\}$, because $\alpha^s_\sim(I_0) = (s \mapsto \top)$. Since the property holds, trying to derive a triple $\vdash_{A^s_\sim} [I_0]\ \mathsf{m}_\varphi\ [\varnothing]$ leads to the failure of some local completeness assumption, which can drive the refinement of A. The imprecision is found in the fourth iteration of the next operator, in which the set $\{(3, 1100), (3, 0111)\}$ is abstracted to $(3, xyzw)$, losing the relation between p and q being either both valid or both invalid. Several domain repairs are possible. Following [8] we can repair the domain by adding the abstract point $q \to p$ (a more abstract repair than adding the element $q \leftrightarrow p$, as proposed in [4]). Here we can also exploit a different route, by refining the equivalence so that $\sigma^\sharp_1 \sim \sigma^\sharp_2$ when $supp(\sigma^\sharp_1) = supp(\sigma^\sharp_2) \neq 3$. This intuitively corresponds to abstract separately the states at program point $n = 3$. In both cases, it holds $[\![\lfloor\overline{\varphi}\rceil]\!]^\sharp \alpha'^s_\sim(I_0) = \bot$ in the refined abstract domain A'. ⌟

▶ **Example 6.5** (Traffic light example, LCL derivation and repair, μ-calculus version)**.** Consider the toy traffic light example from Figure 1a and the property $\psi = \mathsf{AG}\ (\mathsf{g} \to \mathsf{AX}\ \mathsf{d})$, briefly discussed in the introduction. We give some additional details in the light of the results presented. We first translate the formula in $\mu_\Box$-calculus as $\psi_\mu = \nu x.((\mathsf{g} \to \Box\mathsf{d}) \wedge \Box x)$ whose encoding is $\lfloor\overline{\psi_\mu}\rceil = \mu\mathsf{X}.(\mathsf{r}_1 \oplus \mathsf{push}; \mathsf{next}; \mathsf{X}; \mathsf{pop}) \triangleq \mu\mathsf{X}.\mathsf{r}_x$, where $\mathsf{r}_1 \triangleq \mathsf{g}?; \mathsf{push}; \mathsf{next}; \neg\mathsf{d}?; \mathsf{pop}$. Then we try to derive the triple $\vdash_{A^s_\sim} [\{\mathsf{rs}\}]\ \mu\mathsf{X}.\ \mathsf{r}_x[\varnothing]$ in the abstract domain A (see Figure 1c) by applying the rule (afix) with $n = 2$ (the number of iterations needed for convergence of the abstract fixpoint in A). By definition, $\mu^0\mathsf{X}.\mathsf{r}_x = \mathsf{0}$; then the first approximant is $\mu^1\mathsf{X}.\mathsf{r}_x = \mathsf{r}_1 \oplus \mathsf{push}; \mathsf{next}; \mathsf{0}; \mathsf{pop}$, but we can omit the branch that contains $\mathsf{0}$, so $\mu^1\mathsf{X}.\mathsf{r}_x = \mathsf{r}_1$; and the second approximant is $\mathsf{r}_2 \triangleq \mu^2\mathsf{X}.\mathsf{r}_x = \mathsf{r}_1 \oplus \mathsf{push}; \mathsf{next}; \mathsf{r}_1; \mathsf{pop}$. The attempt to derive the triple $\vdash_{A^s_\sim} [\{\mathsf{rs}\}]\ \mathsf{r}_2\ [\varnothing]$ is sketched below, where the mid row gives an indication of the triples labelling the leaves of the derivation, reporting for each basic command involved, the pre-condition and post-condition of the triple, while the top row reports their abstractions.

$$
\begin{array}{cccccccccccc}
 & a & \bot & \bot & \bot & \bot & \bot & a & a & a \vee c & c & c \\
 & [\mathsf{rs}] & [\varnothing] & [\varnothing] & [\varnothing] & [\varnothing] & [\varnothing] & [\mathsf{rs}] & \left[\begin{smallmatrix}\mathsf{rs}\\ \cdots\end{smallmatrix}\right] & \left[\begin{smallmatrix}\mathsf{rs} & \mathsf{gs}\\ \cdots & \cdots\end{smallmatrix}\right] & \left[\begin{smallmatrix}\mathsf{gs}\\ \cdots\end{smallmatrix}\right] & \left[\begin{smallmatrix}\mathsf{gs}\\ \cdots\end{smallmatrix}\right] & \mathbf{!} \\
\vdash_{A^s_\sim} & [\mathsf{rs}] & \mathsf{g}?\,; & \mathsf{push}\,; & \mathsf{next}\,; & \neg\mathsf{d}?\,; & \mathsf{pop} & \oplus & \mathsf{push}\,; & \mathsf{next}\,; & \mathsf{g}?\,; & \mathsf{push}\,; \ \mathsf{next}\,; \ \neg\mathsf{d}?\,; \ \mathsf{pop}\,; \ \mathsf{pop} \ [\varnothing]
\end{array}
$$

While we can derive $\vdash_{A^s_\sim} [\{\mathsf{rs}\}]\ \mathsf{r}_1[\varnothing]$, the source of local incompleteness is found in the second branch, at the **!** position, where the proof obligation for $\mathbb{C}^{A^s_\sim}_{\{\mathsf{gs}\}}(\mathsf{next})$ fails, since $\alpha^s_\sim \circ [\![\mathsf{next}]\!](\{\mathsf{gs}\}) = \alpha^s_\sim(\{\mathsf{gd}, \mathsf{yd}\}) = b \wedge c$, while $\alpha^s_\sim \circ [\![\mathsf{next}]\!] \circ \gamma^s_\sim \circ \alpha^s_\sim(\{\mathsf{gs}\}) = \alpha^s_\sim \circ [\![\mathsf{next}]\!]\ \{\mathsf{gs}, \mathsf{gd}, \mathsf{gs}, \mathsf{ys}\} = \alpha^s_\sim(\{\mathsf{gs}, \mathsf{gd}, \mathsf{gs}, \mathsf{ys}, \mathsf{rs}\}) = a \vee c$. We can thus repair the abstract domain A. Following the procedure in [8], we add the new abstract point c_1:

$$c_1 \triangleq \bigvee \{T \subseteq \Sigma \mid T \subseteq \gamma \circ \alpha\,\{\mathsf{gs}\}\,,\ [\![\mathsf{next}]\!]T \subseteq [\![\mathsf{next}]\!](\{\mathsf{gs}\})\} = \{\mathsf{gs}, \mathsf{gd}\}$$

The repaired domain is $A_1 \triangleq A \cup \{c_1, b \wedge c_1\}$, where $b \wedge c_1$ is also added because the domain must be closed under meets. Now, in A_1, the proof obligation for $\mathbb{C}^{A^s_{1\sim}}_{\{\mathsf{gs}\}}(\mathsf{next})$ holds true: $[\![\mathsf{next}]\!](\{\mathsf{gs}\}) = \alpha_1{}^s_\sim \circ [\![\mathsf{next}]\!] \circ \gamma \circ \alpha_1{}^s_\sim(\{\mathsf{gs}\}) = b \wedge c$. In fact, in A_1 we just have $[\![\lfloor\overline{\psi}\rceil]\!]^\sharp \alpha_1{}^s_\sim(\{\mathsf{rs}\}) = \bot$ (we omit the details for the first branch r_1 that are as above):

$$
\begin{array}{cccccccccccc}
 & a & \bot & a & a & a \vee c & c_1 & c_1 & b \wedge c & \bot & \bot & \bot \\
 & [\mathsf{rs}] & [\varnothing] & [\mathsf{rs}] & \left[\begin{smallmatrix}\mathsf{rs}\\ \cdots\end{smallmatrix}\right] & \left[\begin{smallmatrix}\mathsf{rs} & \mathsf{gs}\\ \cdots & \cdots\end{smallmatrix}\right] & \left[\begin{smallmatrix}\mathsf{gs}\\ \cdots\end{smallmatrix}\right] & \left[\begin{smallmatrix}\mathsf{gs}\\ \cdots\end{smallmatrix}\right] & \left[\begin{smallmatrix}\mathsf{gd} & \mathsf{yd}\\ \cdots & \cdots\end{smallmatrix}\right] & [\varnothing] & [\varnothing] & [\varnothing] \\
\vdash_{A^s_{1\sim}} & [\mathsf{rs}] & \mathsf{r}_1 & \oplus & \mathsf{push}\,; & \mathsf{next}\,; & \mathsf{g}?\,; & \mathsf{push}\,; & \mathsf{next}\,; & \neg\mathsf{d}?\,; & \mathsf{pop}\,; & \mathsf{pop}\ [\varnothing]
\end{array}
$$

⌟

7 Related Work

Abstract interpretation and model checking have been applied to and combined with each other in several ways [2, 4, 5, 10–12, 14, 19–22, 25, 26, 28–30, 33, 35, 37, 38, 44–47, 49, 50] (see the survey [23] for further references). Abstract model checking broadly refers to checking a (temporal) specification in an abstract rather than a concrete model. On the one hand, in state partition-based approaches, the abstract model is itself a transition system, possibly induced by a behavioural state equivalence such as (bi)simulation, so that the verification method for the abstract model remains unaltered, e.g., [10–12, 14, 37]. On the other hand, if the abstract model derives from any, possibly nonpartitioning, state abstraction then the verification on this abstract model relies on abstract interpretation, e.g., [5, 21, 25, 38, 45, 46]. Hybrid abstraction techniques, in between these two approaches, have also been studied, e.g., predicate abstraction [28, 33], and the mixed transition systems with several universal and/or existential abstract transition relations in [3, 22]. Moreover, several works investigated the role of complete/exact/strongly preserving abstractions in model checking, e.g., [20, 26, 37, 44], and how to refine abstract models, e.g., [11, 12, 19, 21, 25, 29, 30, 45, 47, 50]. Let us also mention that [2] develops a theory of approximation for systems of fixpoint equations in the style of abstract interpretation, showing that up-to techniques can be interpreted as abstractions.

Our refinement technique somehow resembles CounterExample-Guided Abstraction Refinement (CEGAR) [12], a popular method for automatising the abstraction generation in model checking. CEGAR deals with state partition abstractions, thus merging sets of equivalent states: one starts from a rough abstraction which is iteratively refined on the basis of spurious counterexample traces arising due to over-approximation. The approach is sound for safety properties, i.e., no false positives can be found, and complete for a significant fragment of ACTL*. It turns out that state partition abstractions are a specific instance in our approach. The attempt of proving the absence of counterexamples in LCL using an initial coarse abstraction will yield a computation by some means similar to CEGAR: failing LCL proof obligations lead to abstraction refinements which can be more general than partitions.

The general idea that model checking can be expressed as static analysis has been first investigated in [40]. This work shows that model checking of ACTL can be reduced to a logic-based static analysis formulated within the flow logic approach [42], which is then computed through a solver for the alternation-free least fixed point logic designed in [41]. This reduction technique has been later extended to the μ-calculus in [55]. One major goal of [40] was to show the close relationship and interplay between model checking and static analysis, coupled with early work in [51] and, later, in [48, 49, 52], proving that static program analysis can be reduced to model checking of modal formulae. Let us remark that the reduction of [40, 55] to a flow logic-based static analysis is given for concrete model checking only and does not encompass the chance of dealing with abstract model checking and related abstraction refinement techniques such as CEGAR, that can be instead achieved by-design in our approach.

8 Conclusion and future work

We have introduced a framework where model checking of temporal formulae in ACTL or in the universal fragment of the modal μ-calculus can be reduced to program verification, paving the way to reuse the full range of abstract interpretation techniques. Formulae are mapped to programs of the MOKA language, that are then analysed through a sound-by-construction abstract interpretation. This exposes all the possible counterexamples, although false alarms can arise. We show how false alarms can be removed by inspecting the derivability of suitable

judgements in LCL, a program logic exploiting under- and over-approximations leveraging the notion of locally complete abstract interpretation. We expect that our approach, relying on KAF, naturally applies also to logics including operators based on regular expressions (see, e.g., [6, 31]). A first candidate is Propositional Dynamic Logic (PDL) [31], a modal logic closely connected to KAT. Its distinctive feature is a modal operator $[\mathsf{r}]\varphi$ where r is a KAT expression, which is satisfied by a state σ when all the computations of r starting from σ end up in a state satisfying φ. For instance, the property expressed in the $\mu_{\Box}$-calculus in Example 4.3 can be equivalently written in PDL as $[\mathsf{next}]^3(n = 3 \to ([\mathsf{next}]^4)^*(n = 3))$, where, since we work in an unlabelled setting, "next" stands for the only action in the system. Indeed, PDL smoothly fits in our setting (we refer the interested reader to [1, Appendix E]).

Future Work. A number of interesting avenues of future research remain open. Our results are limited to universal fragments of temporal logics and finite state domains. A dual theory can be easily developed for existential fragments, focusing on the generation of witnesses rather than counterexamples. It would be interesting to combine the two approaches for dealing with universal and existential operators at the same time. Some ideas could come from [22] that, for solving the problem, works with two different abstract transition relations. The restriction to finite state domains is due to the fact that our encoding of logical formulae into MOKA programs relies on the loop? operator for detecting infinite traces which are identified with looping traces. Further work could overcome this restriction by considering an encoding that captures non-looping infinite traces in the concrete domain and by exploiting ACC domains for the abstraction.

The use of LCL allows us to track the presence of false alarms back to the failure of local completeness proof obligations, which can be resolved by refining the abstract domain. This can be done at different levels: either refining the abstraction over states or refining the equivalence on abstract frames. A proper theory of refinements, possibly identifying optimal ways of patching the domain, is a matter of future investigations.

We point out that providing a general bound for the complexity of the abstract model checking procedure is not straightforward, as it crucially depends on the choice of the abstract domain. Different domains may induce significantly different behaviors, especially for non-partitioning abstractions or when local completeness and domain refinement techniques are applied. A precise complexity analysis tailored to specific domains is an interesting subject for future work.

Concerning the automatisation, the abstract interpreter could be easily implementable leveraging the standard toolset of abstract interpretation. The abstract interpreter should be instrumented to report, when the result is not $\bot$, an abstract counterexample trace to check whether the counterexample is a false or true alarm. Making refinements effective requires working in a class of domains where local refinements are representable, e.g., by predicate abstractions or state partition abstractions. Developing a theory of refinements within specific subclasses of domains is a direction of future work.

References

1 Paolo Baldan, Roberto Bruni, Francesco Ranzato, and Diletta Rigo. Model checking as program verification by abstract interpretation (extended version). *CoRR*, abs/2506.05525, 2025. URL: `https://arxiv.org/abs/2506.05525`.

2 Paolo Baldan, Barbara König, and Tommaso Padoan. Abstraction, up-to techniques and games for systems of fixpoint equations. In *Proceedings of CONCUR 2020*, volume 171 of *LIPIcs*, pages 25:1–25:20, 2020. `doi:10.4230/LIPICS.CONCUR.2020.25`.

3 Thomas Ball, Orna Kupferman, and Greta Yorsh. Abstraction for falsification. In *Proceedings of CAV 2005*, volume 3576 of *LNCS*, pages 67–81. Springer, 2005. doi:10.1007/11513988_8.

4 Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Boolean and cartesian abstraction for model checking C programs. In *Proceedings of TACAS 2001*, volume 2031 of *LNCS*, pages 268–283. Springer, 2001. doi:10.1007/3-540-45319-9_19.

5 Gourinath Banda and John P. Gallagher. Constraint-based abstract semantics for temporal logic: A direct approach to design and implementation. In *Proceedings of LPAR 2016*, volume 6355 of *LNCS*, pages 27–45. Springer, 2010. doi:10.1007/978-3-642-17511-4_3.

6 Ilan Beer, Shoham Ben-David, and Avner Landver. On-the-fly model checking of RCTL formulas. In *Proceedings of CAV 1998*, volume 1427 of *LNCS*, pages 184–194. Springer, 1998. doi:10.1007/BFB0028744.

7 Roberto Bruni, Roberto Giacobazzi, Roberta Gori, and Francesco Ranzato. A logic for locally complete abstract interpretations. In *Proceedings of LICS 2021*. IEEE, 2021. doi:10.1109/LICS52264.2021.9470608.

8 Roberto Bruni, Roberto Giacobazzi, Roberta Gori, and Francesco Ranzato. Abstract interpretation repair. In *Proceedings of PLDI 2022*, pages 426–441. ACM, 2022. doi:10.1145/3519939.3523453.

9 Roberto Bruni, Roberto Giacobazzi, Roberta Gori, and Francesco Ranzato. A correctness and incorrectness program logic. *Journal of the ACM*, 70(2):15:1–15:45, 2023. doi:10.1145/3582267.

10 Doron Bustan and Orna Grumberg. Simulation-based minimization. *ACM Transactions on Computational Logic*, 4(2):181–206, 2003. doi:10.1145/635499.635502.

11 Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Proceedings of CAV 2000*, volume 1855 of *LNCS*, pages 154–169. Springer, 2000. doi:10.1007/10722167_15.

12 Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, 2003. doi:10.1145/876638.876643.

13 Edmund M. Clarke, Orna Grumberg, Daniel Kroening, Doron A. Peled, and Helmut Veith. *Model Checking, 2nd Edition*. MIT Press, 2018. URL: https://mitpress.mit.edu/books/model-checking-second-edition.

14 Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming, Languages and Systems*, 16(5):1512–1542, 1994. doi:10.1145/186025.186051.

15 Edmund M. Clarke, William Klieber, Milos Novácek, and Paolo Zuliani. Model checking and the state explosion problem. In *Tools for Practical Software Verification, LASER, International Summer School 2011*, volume 7682 of *LNCS*, pages 1–30. Springer, 2011. doi:10.1007/978-3-642-35746-6_1.

16 Patrick Cousot. *Principles of Abstract Interpretation*. MIT Press, 2021. URL: https://mitpress.mit.edu/9780262044905/.

17 Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of POPL 1977*, pages 238–252. ACM, 1977. doi:10.1145/512950.512973.

18 Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of POPL 1979*, pages 269–282. ACM Press, 1979. doi:10.1145/567752.567778.

19 Patrick Cousot and Radhia Cousot. Refining model checking by abstract interpretation. *Automated Software Engineering*, 6(1):69–95, 1999. doi:10.1023/A:1008649901864.

20 Patrick Cousot and Radhia Cousot. Temporal abstract interpretation. In *Proceedings of POPL 2000*, pages 12–25. ACM, 2000. doi:10.1145/325694.325699.

21 Patrick Cousot, Pierre Ganty, and Jean-François Raskin. Fixpoint-guided abstraction refinements. In *Proceedings of SAS 2007*, volume 4634 of *LNCS*, pages 333–348. Springer, 2007. doi:10.1007/978-3-540-74061-2_21.

22 Dennis Dams, Rob Gerth, and Orna Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming, Languages and Systems*, 19(2):253–291, 1997. doi:10.1145/244795.244800.

23 Dennis Dams and Orna Grumberg. Abstraction and abstraction refinement. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 385–419. Springer, 2018. doi:10.1007/978-3-319-10575-8_13.

24 Brian A. Davey and Hilary A. Priestley. *Introduction to Lattices and Order, Second Edition*. Cambridge University Press, 2002. doi:10.1017/CBO9780511809088.

25 Roberto Giacobazzi and Elisa Quintarelli. Incompleteness, counterexamples, and refinements in abstract model-checking. In *Proceedings of SAS 2001*, volume 2126 of *LNCS*, pages 356–373. Springer, 2001. doi:10.1007/3-540-47764-0_20.

26 Roberto Giacobazzi and Francesco Ranzato. Incompleteness of states w.r.t. traces in model checking. *Information and Computation*, 204(3):376–407, 2006. doi:10.1016/J.IC.2006.01.001.

27 Roberto Giacobazzi, Francesco Ranzato, and Francesca Scozzari. Making abstract interpretations complete. *Journal of the ACM*, 47(2):361–416, 2000. doi:10.1145/333979.333989.

28 Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In *Proceedings of CAV 1997*, volume 1254 of *LNCS*, pages 72–83. Springer, 1997. doi:10.1007/3-540-63166-6_10.

29 Orna Grumberg, Martin Lange, Martin Leucker, and Sharon Shoham. When not losing is better than winning: Abstraction and refinement for the full mu-calculus. *Information and Computation*, 205(8):1130–1148, 2007. doi:10.1016/J.IC.2006.10.009.

30 Anubhav Gupta and Ofer Strichman. Abstraction refinement for bounded model checking. In *Proceedings of CAV 2005*, volume 3576 of *LNCS*, pages 112–124. Springer, 2005. doi:10.1007/11513988_11.

31 David Harel, Jerzy Tiuryn, and Dexter Kozen. *Dynamic Logic*. MIT Press, Cambridge, MA, USA, 2000. URL: https://mitpress.mit.edu/9780262527668/.

32 Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Computing Surveys*, 41(4):21:1–21:54, 2009. doi:10.1145/1592434.1592438.

33 Ranjit Jhala, Andreas Podelski, and Andrey Rybalchenko. Predicate abstraction for program verification. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 447–491. Springer, 2018. doi:10.1007/978-3-319-10575-8_15.

34 Dexter Kozen. Kleene algebra with tests. *ACM Transactions on Programming, Languages and Systems*, 19(3):427–443, 1997. doi:10.1145/256167.256195.

35 Kim Guldstrand Larsen and Bent Thomsen. A modal process logic. In *Proceedings of LICS 1988*, pages 203–210. IEEE Computer Society, 1988. doi:10.1109/LICS.1988.5119.

36 Hans Leiß. Towards Kleene algebra with recursion. In *Proceedings of CSL 1991*, volume 626 of *LNCS*, pages 242–256. Springer, 1991. doi:10.1007/BFB0023771.

37 Claire Loiseaux, Susanne Graf, Joseph Sifakis, Ahmed Bouajjani, and Saddek Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6(1):11–44, 1995. doi:10.1007/BF01384313.

38 Damien Massé. Semantics for abstract interpretation-based static analyzes of temporal properties. In *Proceedings of SAS 2002*, volume 2477 of *LNCS*, pages 428–443. Springer, 2002. doi:10.1007/3-540-45789-5_30.

39 Antoine Miné. Tutorial on static inference of numeric invariants by abstract interpretation. *Foundations and Trends in Programming Languages*, 4(3-4):120–372, 2017. doi:10.1561/2500000034.

40 Flemming Nielson and Hanne Riis Nielson. Model checking *is* static analysis of modal logic. In *Proceedings of FoSSaCS 2010*, volume 6014 of *LNCS*, pages 191–205. Springer, 2010. doi:10.1007/978-3-642-12032-9_14.

41 Flemming Nielson, Helmut Seidl, and Hanne Riis Nielson. A succinct solver for ALFP. *Nordic Journal of Computing*, 9(4):335–372, 2002.

42 Hanne Riis Nielson and Flemming Nielson. Flow logic: A multi-paradigmatic approach to static analysis. In *The Essence of Computation, Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones [on occasion of his 60th birthday]*, volume 2566 of *LNCS*, pages 223–244. Springer, 2002. doi:10.1007/3-540-36377-7_11.

43 Peter W. O'Hearn. Incorrectness logic. *Proceedings of POPL 2020*, 4:10:1–10:32, 2020. doi:10.1145/3371078.

44 Francesco Ranzato. On the completeness of model checking. In *Proceedings of ESOP 2001*, volume 2028 of *LNCS*, pages 137–154. Springer, 2001. doi:10.1007/3-540-45309-1_10.

45 Francesco Ranzato and Francesco Tapparo. Making abstract model checking strongly preserving. In *Proceedings of SAS 2002*, volume 2477 of *LNCS*, pages 411–427. Springer, 2002. doi:10.1007/3-540-45789-5_29.

46 Francesco Ranzato and Francesco Tapparo. Generalized strong preservation by abstract interpretation. *Journal of Logic and Computation*, 17(1):157–197, 2007. doi:10.1093/LOGCOM/EXL035.

47 David Schmidt. Binary relations for abstraction and refinement. Technical report, Kansas State University, 2001.

48 David A. Schmidt. Data flow analysis is model checking of abstract interpretations. In *Proceedings of POPL 1998*, pages 38–48. ACM, 1998. doi:10.1145/268946.268950.

49 David A. Schmidt and Bernhard Steffen. Program analysis *as* model checking of abstract interpretations. In *Proceedings of SAS 1998*, volume 1503 of *LNCS*, pages 351–380. Springer, 1998. doi:10.1007/3-540-49727-7_22.

50 Sharon Shoham and Orna Grumberg. Monotonic abstraction-refinement for CTL. In *Proceedings of TACAS 2004*, volume 2988 of *LNCS*, pages 546–560. Springer, 2004. doi:10.1007/978-3-540-24730-2_40.

51 Bernhard Steffen. Data flow analysis as model checking. In *Proceedings of TACS 1991*, volume 526 of *LNCS*, pages 346–365. Springer, 1991. doi:10.1007/3-540-54415-1_54.

52 Bernhard Steffen. Generating data flow analysis algorithms from modal specifications. *Science of Computer Programming*, 21(2):115–139, 1993. doi:10.1016/0167-6423(93)90003-8.

53 Colin Stirling and David Walker. Local model checking in the modal mu-calculus. In *Proceedings of TAPSOFT 1989*, volume 351 of *LNCS*, pages 369–383. Springer, 1989. doi:10.1007/3-540-50939-9_144.

54 Pierre Wolper. Temporal logic can be more expressive. *Information and Control*, 56(1/2):72–99, 1983. doi:10.1016/S0019-9958(83)80051-5.

55 Fuyuan Zhang, Flemming Nielson, and Hanne Riis Nielson. Model checking as static analysis: Revisited. In *Proceedings of IFM 2012*, volume 7321 of *LNCS*, pages 99–112. Springer, 2012. doi:10.1007/978-3-642-30729-4_8.

A Direct Reduction from Stochastic Parity Games to Simple Stochastic Games

Raphaël Berthon
RWTH Aachen University, Germany

Joost-Pieter Katoen
RWTH Aachen University, Germany

Zihan Zhou
National University of Singapore, Singapore

Abstract

Significant progress has been recently achieved in developing efficient solutions for simple stochastic games (SSGs), focusing on reachability objectives. While reductions from stochastic parity games (SPGs) to SSGs have been presented in the literature through the use of multiple intermediate game models, a direct and simple reduction has been notably absent. This paper introduces a novel and direct polynomial-time reduction from quantitative SPGs to quantitative SSGs. By leveraging a gadget-based transformation that effectively removes the priority function, we construct an SSG that simulates the behavior of a given SPG. We formally establish the correctness of our direct reduction. Furthermore, we demonstrate that under binary encoding this reduction is polynomial, thereby directly corroborating the known $\mathbf{NP} \cap \mathbf{coNP}$ complexity of SPGs and providing new understanding in the relationship between parity and reachability objectives in turn-based stochastic games.

2012 ACM Subject Classification Theory of computation → Probabilistic computation

Keywords and phrases stochastic games, parity, reduction

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2025.9

Related Version *Full Version*: `https://arxiv.org/abs/2506.06223` [6]

Funding *Raphaël Berthon*: Funded by DFG Project POMPOM (KA 1462/6-1).

1 Introduction

Stochastic games (SGs) are a broadly used framework for decision-making under uncertainty with a well-defined objective. They aim at modeling two important aspects under which sequential decisions must be made: turn-based interaction with an adversary environment, and dealing with randomness. Stochastic games have been introduced long ago by Shapley [40] as important models, and many variations have been shown to be inter-reducible and in $\mathbf{NP} \cap \mathbf{coNP}$ [18, 2, 9]. SGs find their applications in various fields, including artificial intelligence [33], economics [1], operations research [20], or providing tools to graph theory [41]. Moreover, *Markov Decision Processes* (MDP) [38], a foundational framework for modeling decision-making in stochastic environments, are special cases of SGs, where one of the players has no states under control.

While we make use of simple stochastic games (SSGs) with reachability objectives, we focus on the specific case of stochastic *parity* games (SPGs), which are zero-sum, and where the set of winning runs is ω-regular. Solving such a game consists in finding an optimal strategy and determining its winning probability.

Take, for instance, the case of a market competition, where two firms Alpha and Beta try to expand their market share. From the standpoint of each firm, the other acts as a direct competitor, and therefore we assume the players are *adversarial*. States represent the relative valuation of the firms over a sustained period of time, and business decisions

© Raphaël Berthon, Joost-Pieter Katoen, and Zihan Zhou;
licensed under Creative Commons License CC-BY 4.0

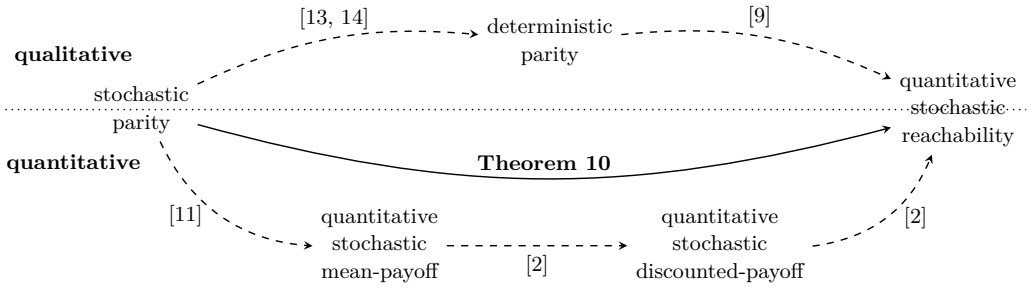


Figure 1 Reducing SPGs to SSGs.

(made by either firm Alpha or Beta) and the fluctuation of market shares, policy changes, and external forces (modeled as randomness) lead to transitions between game states. Firm Alpha is interested in keeping its share above a set key threshold, say 50%. We distinguish between three priorities, leading to a *zero-sum* game:

- *Priority* 0: Alpha's market share is significantly above 50%.
- *Priority* 1: Alpha's market share is significantly below 50%.
- *Priority* 2: Alpha's market share fluctuates around 50%. While this is sustainable without major fluctuation, this is not sustainable if the only other fluctuation is Alpha's share regularly dropping below 50% (priority 1).

If the minimum priority visited infinitely often is 0, Alpha can manage in the long term to regularly dominate the market, recovering any loss that occurred in the meantime. If the minimum priority visited infinitely often is 1, despite any temporary success, Alpha's market share will stay near or below 50% in the long run, which is not sustainable for the firm. Finally, if the minimum priority visited infinitely often is 2, Alpha and Beta will eventually find an even balance point.

A variety of algorithms have been considered for (reachability) SSGs [17, 21, 3, 37], which we present in our related work section below. In Markov chains (MCs), ω-regular objectives can be reduced directly to reachability objectives [4]. A similar reduction exists from MDPs with ω-regular objectives to reachability objectives and has been used extensively, for example in [22]. This means that solvers often focus on optimizing specifically the computation of reachability probabilities. Such a direct reduction is lacking for SPGs. To reduce quantitative SPGs to SSGs, some intermediate steps are necessary, via a reduction to stochastic mean-payoff and stochastic discounted-payoff games [11, 2] (see the lower part of Figure 1), making this approach less appealing. For qualitative solutions, a translation via deterministic parity games (i.e. with no random states) exists [13, 14, 9], see the upper part of Figure 1.

Outline and Contribution

In this paper, we propose a *direct* reduction from SPGs to SSGs with a reachability objective (see the solid arrow in Figure 1). To that end, we leverage a gadget whose structure comes from [9], but where we use new probability values, to reduce deterministic parity games to quantitative SSGs. Given an SPG G, where a parity condition is satisfied if the minimum priority seen infinitely often is even, we show in Section 3.1 how to use the gadget to transform G into an SSG $\widetilde{G}$. This introduces two new sink states, one winning and the other losing for the reachability objective. Parity values are removed, and every transition going to a state that used to be even (respectively odd) now has a small chance to go to the winning (resp.

losing) sink. We scale the probabilities, with lower parity values yielding a higher chance to go to a sink. Theorem 10 ensures that any optimal strategy in $\widetilde{G}$ is also optimal in G (the reciprocal may not be true). We can then compute an optimal memoryless strategy in $\widetilde{G}$, and compute its value in G.

We show in Theorem 13 that under binary encoding, our reduction is polynomial. We thus reobtain in a direct way the classical result that solving quantitative SPGs is in the complexity class $\mathbf{NP} \cap \mathbf{coNP}$ [2, 9]. While the complexity remains the same as for existing algorithms, and the values used in the reduction make it unlikely to be very efficient in practice, this new approach implies that any efficient SSG solver can be used for SPGs. The direct reduction was already conjectured to exist by Chatterjee and Fijalkow in [9], but as expected, proving its correctness is challenging, and involves the computations of very precise probability bounds. Despite the inspiration drawn from a known gadget, the technical depth of this paper resides in the intricate and novel proofs for the correctness of our reduction. In addition, our direct reduction gives new insights into the relationship between SSGs and SPGs.

In Section 2, we present the necessary background knowledge. In Section 3, we define the gadget and present some related results, which we use in Section 4 to define the reduction properly, and to show its correctness and its complexity. We give some concluding remarks in Section 5.

Related Work

Stochastic parity games, mean-payoff games and discounted payoff games can all be reduced to SSGs [26, 43], and this also applies to their stochastic extensions, namely stochastic parity games [9], stochastic mean payoff games and stochastic discounted payoff games [2]. SSGs also find their applications in the analysis of MDPs, serving as abstractions for large MDPs [28]. The amount of memory required to solve stochastic parity games has been studied in [7].

Various extensions have been considered within this family of inter-reducible stochastic games. Introducing more than two players allows for the analysis of Nash equilibria [15, 42]. Using continuous states can provide tools to represent timed systems [34]. Multi-objective approaches have been employed to synthesize systems that balance average expected outcomes with worst-case guarantees [16]. Parity objectives are significant in many of these scenarios where long-run behavior is relevant, but the classical reduction to SSGs cannot be directly applied.

Common approaches to solving SSGs, as presented in [17], include value iteration (VI), strategy iteration (SI), and quadratic programming, but are all exponential in the size of the SSG. These approaches have been widely studied on MDPs, where recent advancements have been made to apply VI with guarantees, using interval VI [5], sound VI [39], and optimistic VI [24]. Interestingly, optimistic VI does not require an a priori computation of starting vectors to approximate from above. Similar ideas have been lifted to SSGs: Eisentraut et al. [21] introduce a VI algorithm for under- and over-approximation sequences, as well as the first practical stopping criterion for VI on SSGs. Optimistic VI has been adapted to SSGs [3], and a novel bounded VI with the concept of widest path has been introduced in [37]. A comparative analysis [29] suggests VI and SI are more efficient. Storm [25] and PRISM [31] are two popular model checkers incorporating different variants of VI and SI, and both employ VI as the default algorithm for solving MDPs. PRISM-games [30] exploits VI for solving SSGs.

For SPGs, we distinguish three main approaches. Chatterjee et al. [10] use a strategy improvement algorithm requiring randomized sub-exponential time. With n game states and d priorities, the expected running time is in $2^{O(\sqrt{dn\log(n)})}$. The probabilistic game solver GIST [12] reduces qualitative SPGs to deterministic parity games (DPG), and benefits from several quasi-polynomial algorithms for DPGs [27, 36, 32] since the breakthrough made by Calude et al. [8], but this approach is unlikely to achieve polynomial running time [19]. Hahn et al. [23] reduce SPGs to SSGs, allowing the use of reinforcement learning to approximate the values without knowing the game's probabilistic transition structure. Their reduction is only proven correct in the limit.

2 Preliminaries

Our notations on Markov chains and stochastic games on graphs mainly come from [4].

2.1 Discrete-Time Markov Chains

A *discrete distribution* over a countable set $\mathcal{A}$ is a function $\mu : \mathcal{A} \to \mathbb{R}_{\geq 0}$ with $\sum_{a\in\mathcal{A}} \mu(a) = 1$. The *support* of the discrete distribution μ is defined as $supp(\mu) \triangleq \{a \in \mathcal{A} \mid \mu(a) > 0\}$. We denote the set of all discrete distributions over $\mathcal{A}$ with $\mathbb{D}(\mathcal{A})$.

A *discrete-time Markov Chain (MC)* $\mathcal{M}$ is a tuple $\mathcal{M} = (V, \delta, v_I)$ where V is a finite set of states, $\delta : V \to \mathbb{D}(V)$ is a probabilistic transition function, and $v_I \in V$ is the initial state. Given $\delta(v) = \mu$ with $\mu(v') = p$, we write $\delta(v, v') = p$. For $S \subseteq V$ and $v \in V$, let $\delta(v, S) = \sum_{s\in S} \delta(v, S)$.

An infinite sequence $\pi = v_0 v_1 \cdots \in V^\omega$ is an *infinite path* through MC $\mathcal{M}$ if $\delta(v_i, v_{i+1}) > 0$ for all $i \in \mathbb{N}$. We denote all infinite paths that start from state $v \in V$ with $Paths(v)$. *Prefixes* of infinite path $\pi = v_0 v_1 \cdots \in V^\omega$ are $\{v_0 \cdots v_i \mid i \in \mathbb{N}\}$ and are *finite* paths. We denote all finite paths that start from state $v \in V$ with $Paths^*(v)$. The set of infinitely often visited states in $\pi = v_0 v_1 \cdots \in V^\omega$ is defined as $inf(\pi) = \{v \in V \mid \forall n \in \mathbb{N}, \exists k \in \mathbb{N} \ s.t. \ v_{n+k} = v\}$.

The probability Pr of a finite path $\pi = v_0 v_1 \dots v_n \in V^*$ is given by $\prod_{i\in[0,n-1]} \delta(v_i, v_{i+1})$. The set of infinite paths that start with a given finite path is called a *cylinder*, and as in [4], we extend the probability of cylinders in a unique way to all *measurable sets* of V^ω.

Reachability Probabilities

Let $\mathcal{M} = (V, \delta, v_I)$ be an MC. For target states $T \subseteq V$ and starting state $v_0 \in V$, the event of reaching T is defined as $Reach(T) = \{v_0 v_1 \cdots \in V^\omega \mid \exists i \in \mathbb{N}, v_i \in T\}$. The probability to reach T from v_0 is defined as $\Pr^{v_0}(Reach(T)) = \Pr(\{\hat{\pi} \mid \hat{\pi} \in Paths^*(v_0) \cap ((V\backslash T)^* T)\})$.

Let variable x_v denote the probability of reaching T from any $v \in V$. Whether T is reachable from a given state v can be determined using standard graph analysis. Let $Pre^*(T)$ denote the set of states from which T is reachable. If $v \notin Pre^*(T)$, then $x_v = 0$. If $v \in T$, then $x_v = 1$. Otherwise, $x_v = \sum_{u\in V\backslash T} \delta(v, u) \cdot x_u + \sum_{w\in T} \delta(v, w)$. This is equivalent to a linear equation system, formalized as follows:

▶ **Theorem 1** (Reachability Probability of Markov Chains [4]). *Given MC $\mathcal{M} = (V, \delta, v_I)$ and target states $T \subseteq V$, let $V_Q = Pre^*(T)\backslash T$, $\mathbf{A} = (\delta(v, v'))_{v,v'\in V_Q}$ and $\mathbf{b} = (b_v)_{v\in V_Q} = (\delta(v, T))_{v\in V_Q}$. Then, the vector $\mathbf{x} = (x_v)_{v\in V_Q}$ with $x_v = \Pr^v(Reach(T))$ is the **unique** solution of the linear equation system $\mathbf{x} = \mathbf{A} \cdot \mathbf{x} + \mathbf{b}$.*

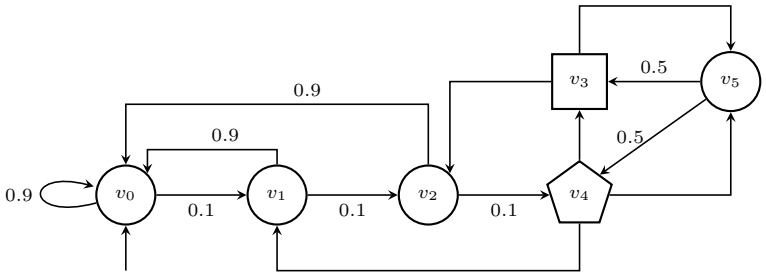


Figure 2 An example stochastic arena G.

Limit Behavior

Let MC $\mathcal{M} = (V, \delta, v_I)$. A set $L \subseteq V$ is *strongly connected* if for all pairs of states $v, v' \in L$, v and v' are mutually reachable. Hence a singleton $\{v\}$ is strongly connected if $\delta(v, v) > 0$. Set $L \subseteq V$ is a *strongly connected component (SCC)* if it is maximally strongly connected, i.e., there does not exist another set $L' \subseteq V$ and $L \subsetneq L'$ such that L' is strongly connected. $L \subseteq V$ is a *bottom SCC (BSCC)* if L is a SCC and there is no transition leaving L, i.e., there does not exist $v \in L, v' \in V \setminus L$ such that $\delta(v, v') > 0$. We denote the set of BSCCs in MC $\mathcal{M}$ with $BSCC(\mathcal{M})$.

The limit behavior of an MC regarding the infinitely often visited states is captured by the following theorem.

▶ **Theorem 2** (Limit behavior of Markov Chains [4]). *For MC $\mathcal{M} = (V, \delta, v_I)$, it holds that* $\Pr\{\pi \in Paths(v_I) \mid inf(\pi) \in BSCC(\mathcal{M})\} = 1$.

2.2 Stochastic Games

A *stochastic arena* G is a tuple $G = ((V, E), (V_\exists, V_\forall, V_R), \Delta)$, where (V, E) is a directed graph, with a finite set of vertices V, partitioned as $V_\exists \uplus V_\forall \uplus V_R = V$, and a set of edges $E \subseteq V \times V$. The probabilistic transition function Δ is such that for all $v_r \in V_R$, $\Delta(v_r)$ is a distribution over V, and for $v \in V_\exists \uplus V_\forall$, v', we have $(v_r, v) \in E$ if and only if $v \in supp(\Delta(v_r))$. We usually uncurry $\Delta(v_r)(v)$ and write $\Delta(v_r, v)$.

Without loss of generality, we assume each vertex has at least one successor. This property is called *non-blocking*. The finite set V of vertices is partitioned into three sets: $V_\exists$ – vertices where Eve chooses the successor, $V_\forall$ – vertices where Adam chooses the successor, and V_R are the random vertices. A stochastic arena is a Markov Decision Process (MDP) if either $V_\exists = \emptyset$ or $V_\forall = \emptyset$, and an MC if both $V_\exists = \emptyset$ and $V_\forall = \emptyset$.

Figure 2 illustrates a stochastic arena G. Square-shaped vertex v_3 is a vertex in $V_\exists$ where Eve chooses the successor, pentagon-shaped vertex v_4 is in $V_\forall$ where Adam chooses the successor, and the circular vertices $V_R = \{v_0, v_1, v_2, v_5\}$ are random. Edges from random vertices are annotated with probabilities from Δ.

Strategies

Let $G = ((V, E), (V_\exists, V_\forall, V_R), \Delta)$ be a stochastic arena. A *strategy* σ of Eve is a function $\sigma : V^* \cdot V_\exists \to \mathbb{D}(V)$, such that for all $v_0 v_1 \dots v_n \in V^* \cdot V_\exists$, we have $\sigma(v_0 v_1 \dots v_n, v_{n+1}) > 0$ implies $(v_n, v_{n+1}) \in E$. A strategy γ of Adam is defined analogously. We denote the sets of all strategies of Eve and Adam with $\Sigma^A_\exists$ and $\Sigma^A_\forall$ respectively.

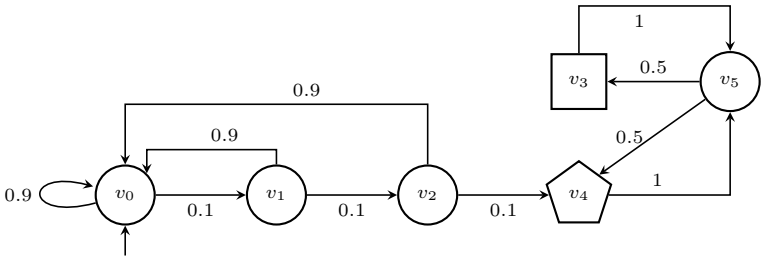


Figure 3 Sub-arena $G_{\sigma,\gamma}$ induced by strategies $\sigma = [v_4 \mapsto v_5]$ and $\gamma = [v_3 \mapsto v_5]$.

A strategy σ of Eve is a *pure memoryless strategy*, if for all $w, w' \in V^*$ and $v \in V_\exists$, $\sigma(w \cdot v) = \sigma(w' \cdot v)$ and the support of this distribution is a singleton. A pure memoryless strategy γ of Adam is defined analogously. We denote the sets of pure memoryless strategies of Eve and Adam with $\Sigma_\exists$ and $\Sigma_\forall$ respectively.

In a stochastic arena G, when Eve and Adam follow pure memoryless strategies $\sigma \in \Sigma_\exists$ and $\gamma \in \Sigma_\forall$ respectively, the arena $G_{\sigma,\gamma} = ((V, E'), (V_\exists, V_\forall, V_R), \Delta)$ results. Here, the new edge set E' is such that for all $u \in V_\exists$, $(u, v) \in E'$ if and only if $\sigma(u) = v$, and for all $u \in V_\forall$, $(u, v) \in E'$ if and only if $\gamma(u) = v$. We refer to such arenas obtained by fixing pure memoryless strategies as sub-arenas. In fact, given a fixed starting vertex $v_I \in V$, we often view the sub-arena $G_{\sigma,\gamma}$ as an MC $\mathcal{M}_{\sigma,\gamma} = (V, \delta, v_I)$, where the state space is the vertex set V in G, and the transition function δ combines deterministic moves indicated by strategies σ and τ, and the transition function Δ defined on random vertices:

$$\delta(u, v) = \begin{cases} \Delta(u, v) & \text{if } u \in V_R, \text{or } u \in V_\exists, \sigma(u) = v \text{ or } u \in V_\forall, \gamma(u) = v \\ 0 & \text{otherwise} \end{cases}$$

We continue with the stochastic arena G from Figure 2. Fixing strategy $\sigma = [v_3 \mapsto v_5]$ for Eve and $\gamma = [v_4 \mapsto v_5]$ for Adam induces the sub-arena $G_{\sigma,\gamma}$, as shown in Figure 3.

Winning Objectives

A *play* of G is an infinite sequence of vertices $\pi = v_0 v_1 \cdots \in V^\omega$ where for all $i \in \mathbb{N}$, $(v_i, v_{i+1}) \in E$. We denote the set of all plays of G with Π_G, or in short Π when G is clear from the context.

Let G be a stochastic arena. A *winning objective* for Eve is defined as a set of plays $\mathcal{O} \subseteq \Pi$. As we study zero-sum games, the winning objectives of the two players are complementary. The winning objective for Adam is thus $\Pi \backslash \mathcal{O}$. A play π *satisfies* an objective $\mathcal{O}$ if $\pi \in \mathcal{O}$, and is a *winning play* of Eve. A winning play π of Adam satisfies $\Pi \backslash \mathcal{O}$.

A *reachability* objective asserts that the play in G has to reach target vertices $T \subseteq V$, formally given by $RE(T) = \{v_0 v_1 \cdots \in \Pi \mid \exists k \in \mathbb{N}, v_k \in T\}$. If $T = \{v\}$ for some vertex v, we simply write $RE(v)$.

Let $p : V \to \mathbb{N}$ be a *priority function* which assigns a priority $p(v)$ to each vertex $v \in V$. For $T \subseteq V$, let $p(T) = \{p(t) \mid t \in T\}$. A parity objective asserts that the minimum priority visited infinitely often along an infinite path is even: $PA(p) = \{\pi = v_0 v_1 \cdots \in \Pi \mid \min(p(\mathit{inf}(\pi)))$ is even$\}$.

We formally define stochastic games as follows:

▶ **Definition 3** (Stochastic Games). *Let $G = ((V, E), (V_\exists, V_\forall, V_R), \Delta)$ be a stochastic arena. A **stochastic game (SG)** with winning objective $\Phi \subseteq \Pi$ is defined as (G, Φ). If Φ is a reachability or parity objective, (G, Φ) is a stochastic reachability game (SRG) or **stochastic***

parity game (SPG) *respectively. SRGs are also referred to as* ***simple stochastic games (SSG)****. When the winning objective is clear from the context, we refer to G as a stochastic game.*

Solving stochastic games

Let (G, Φ) be an SG, and let Eve and Adam follow strategies $\sigma \in \Sigma^A_\exists$ and $\gamma \in \Sigma^A_\forall$. Given a starting vertex $v \in V$, the probability for play π to satisfy Φ – the probability for Eve to win – is denoted $\mathbb{P}^v_{\sigma,\gamma}(\Phi)$. The probability for Adam to win is $\mathbb{P}^v_{\sigma,\gamma}(\Pi\backslash\Phi)$.

Let the *value* of a vertex v be the maximal probability of generating a play from v that satisfies Φ, formally defined using a *value function* $\langle E\rangle(\Phi)(v) = \sup_{\sigma\in\Sigma^A_\exists} \inf_{\gamma\in\Sigma^A_\forall} \mathbb{P}^v_{\sigma,\gamma}(\Phi)$ for Eve, and $\langle A\rangle(\Pi\backslash\Phi)(v) = \sup_{\gamma\in\Sigma^A_\forall} \inf_{\sigma\in\Sigma^A_\exists} \mathbb{P}^v_{\sigma,\gamma}(\Pi\backslash\Phi)$ for Adam. A strategy σ for Eve is *optimal* from vertex v if $\inf_{\gamma\in\Sigma^A_\forall} \mathbb{P}^v_{\sigma,\gamma}(\Phi) = \langle E\rangle(\Phi)(v)$. Optimal strategies for Adam are defined analogously.

We divide solving stochastic games into three distinct tasks. Given an SG, solving the SG *quantitatively* amounts to computing the values of all vertices in the arena. Solving the SG *strategically* amounts to computing an optimal strategy of Eve (or Adam) for the game.

Since for both SSGs and SPGs, solving quantitatively and strategically is polynomially equivalent [2], we just say "solving" in what follows. We mainly consider quantitative solving, but Theorem 10 applies to both quantitative and strategic solving.

Determinacy

Determinacy refers to the property of an SG where both players, Eve and Adam, have optimal strategies, meaning they can guarantee to achieve the values of the game, regardless of the strategies employed by the other player. *Pure memoryless determinacy* means that both players have pure memoryless optimal strategies.

▶ **Theorem 4** (Pure Memoryless Determinacy [35]). *Let (G, Φ) be an SG, where Φ is a reachability or parity objective. For all $v \in V$, it holds that $\langle E\rangle(\Phi)(v) + \langle A\rangle(\Pi\backslash\Phi)(v) = 1$. Pure memoryless optimal strategies exist for both players from all vertices.*

When Eve and Adam follow pure memoryless strategies $\sigma \in \Sigma_\exists$ and $\gamma \in \Sigma_\forall$ respectively, we obtain sub-arena $G_{\sigma,\gamma}$, which can be seen as the MC $\mathcal{M}_{\sigma,\gamma}$. We can reduce the winning probabilities $\mathbb{P}^{v_I}_{\sigma,\gamma}$ to reachability probabilities in $G_{\sigma,\gamma}$ as follows. Given a reachability objective $RE(T)$, $\mathbb{P}^{v_I}_{\sigma,\gamma}(RE(T)) = \Pr^{v_I}_{\sigma,\gamma}(Reach(T))$. Given a parity objective $PA(p)$, , i.e. $(G, PA(p))$, we call $B \in BSCC(\mathcal{M}_{\sigma,\gamma})$ an *even BSCC* if $min(p(B))$ is even, meaning intuitively the smallest priority of its vertices is even. *Odd BSCCs* are defined analogously. Then $\mathbb{P}^{v_I}_{\sigma,\gamma}(PA(p)) = \Pr^{v_I}_{\sigma,\gamma}(Reach(B_E))$, where $B_E = \bigcup_{min(p(B)) \text{ is even}} B \in BSCC(\mathcal{M}_{\sigma,\gamma})$.

▶ **Corollary 5** (Sufficiency of Pure Memoryless Strategies [11]). *Let $G = ((V, E), (V_\exists, V_\forall, V_R), \Delta)$ be a stochastic arena, $RE(T)$ a reachability objective, and $PA(p)$ a parity objective. For all vertices $v \in V$, it holds:*

- $\langle E\rangle(RE(T))(v) = \sup\limits_{\sigma\in\Sigma^A_\exists} \inf\limits_{\gamma\in\Sigma^A_\forall} \mathbb{P}^v_{\sigma,\gamma}(RE(T)) = \sup\limits_{\sigma\in\Sigma_\exists} \inf\limits_{\gamma\in\Sigma_\forall} \Pr^v_{\sigma,\gamma}(Reach(T))$
- $\langle E\rangle(PA(p))(v) = \sup\limits_{\sigma\in\Sigma^A_\exists} \inf\limits_{\gamma\in\Sigma^A_\forall} \mathbb{P}^v_{\sigma,\gamma}(PA(p)) = \sup\limits_{\sigma\in\Sigma_\exists} \inf\limits_{\gamma\in\Sigma_\forall} \Pr^v_{\sigma,\gamma}(Reach(B_E))$

  where $B_E = \bigcup_{min(p(B)) \text{ is even}} B \in BSCC(\mathcal{M}_{\sigma,\gamma})$.

Therefore we consider only pure memoryless strategies in the sequel, unless stated otherwise.

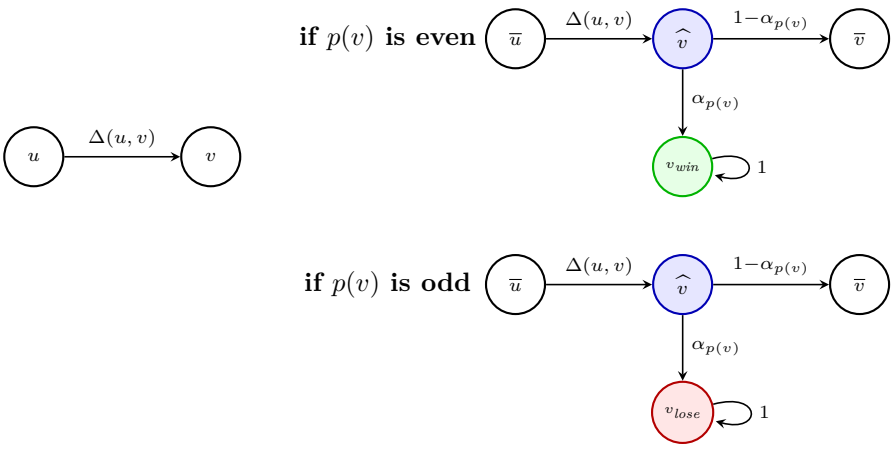


Figure 4 The gadgets for reducing SPG G (left) to SSG $\widetilde{G}$ (right).

3 A Gadget for Transforming SPGs into SSGs

The aim of this paper is to reduce an SPG $(G, PA(p))$ to an SSG $(\widetilde{G}, RE(v_{win}))$ such that the probability of reaching target vertex v_{win} in this SSG is related to the probability of winning in the SPG $(G, PA(p))$. As an important step toward this goal, we introduce in this section a gadget that expands each transition of G while removing the priority function.

Let $G = ((V, E), (V_\exists, V_\forall, V_R), \Delta)$ be a stochastic arena, $p : V \to \mathbb{N}$ be a priority function, and $(G, PA(p))$ be an SPG. Section 3.1 presents the gadget enabling the reduction from SPG to SSG $(\widetilde{G}, RE(v_{win}))$. We then analyze how probabilistic events in $\widetilde{G}$ are related to those in G. Section 3.2 presents a bound on the probability of reaching BSCCs in $\widetilde{G}$. Section 3.3 provides a bound on the winning probability once a BSCC in $\widetilde{G}$ is reached, while Section 3.4 gives interval bounds on the winning probabilities in $\widetilde{G}$ with regard to those in G.

3.1 Gadget Construction

To reduce the parity objective to a reachability objective, we transform the SPG $(G, PA(p))$ into the SSG $(\widetilde{G}, RE(v_{win}))$ by means of a gadget, whose structure was defined by Chatterjee and Fijalkow in [9] to reduce deterministic parity games (DPG) to SSGs.

When both players' strategies are fixed in a DPG, every play forms a lasso, *i.e.* a finite simple path that ends in a simple cycle. Chatterjee and Fijalkow's analysis proceeds by examining lasso plays to choose transition probabilities of the gadget. The specific probabilities they introduce do not extend to a reduction from SPGs to SSGs, because in our stochastic setting, a lasso lifts to a more complicated structure, namely an induced Markov chain: the simple path becomes the transient segment, and each cycle becomes a BSCC. Hence, our approach makes use of smaller probability values, requiring more representation space.

The structure of the gadget remains applicable, and its intuition is as follows: whenever a play visits a vertex with even priority in G, give a small but positive chance to reach a winning sink in $\widetilde{G}$. Vertices with odd priority yield a small chance to reach a losing sink. Finally, to represent that smaller priorities have precedence over larger ones, the probability of reaching a sink from a vertex depends on the priority it is associated to. We introduce a monotonically decreasing function α for this purpose.

We obtain the stochastic arena $\widetilde{G}$ by modifying G as indicated in Figure 4. Each vertex v in G is duplicated in $\widetilde{G}$ yielding vertices $\widehat{v}$ and $\overline{v}$. A transition $\Delta(u,v)$ in G is replaced by first moving to $\widehat{v}$, which can then either evolve to a sink with probability $\alpha_{p(v)}$, or to the copy $\overline{v}$ with the complementary probability. Depending on p being even or odd, the sink is v_{win} or v_{lose}.

Formally, for $U \subseteq V$, let $\overline{U} = \{\overline{v} \mid v \in U\}$, $\widehat{U} = \{\widehat{v} \mid v \in U\}$ and $\widetilde{U} = \overline{U} \uplus \widehat{U}$. We define the arena $\widetilde{G} = ((\widetilde{V} \uplus \{v_{win}, v_{lose}\}, \widetilde{E}), (\overline{V}_\exists, \overline{V}_\forall, \overline{V}_R \uplus \widehat{V} \uplus \{v_{win}, v_{lose}\}), \widetilde{\Delta})$ where the new edge set $\widetilde{E}$ is as follows: $\widetilde{E} = \{(\overline{u}, \widehat{v}) \mid (u,v) \in E\} \uplus \{(\widehat{v}, \overline{v}), (\widehat{v}, v_{win}) \mid v \in V,\ p(v) \text{ is even}\} \uplus \{(\widehat{v}, \overline{v}), (\widehat{v}, v_{lose}) \mid v \in V,\ p(v) \text{ is odd}\} \uplus \{(v_{win}, v_{win}), (v_{lose}, v_{lose})\}$.

To define the new transition function $\widetilde{\Delta}$, let $\alpha : \mathbb{N} \to [0,1]$ where α_i represents the probability of entering the winning (resp. losing) sink before visiting a vertex with even (resp. odd) priority i. We give suitable values for α later, in Lemma 11 on page 14. Now, we define $\widetilde{\Delta} : \widetilde{V} \times \widetilde{V} \to [0,1]$ as follows:

$$\widetilde{\Delta}(\widetilde{u}, \widetilde{w}) = \begin{cases} \Delta(u,w) & \text{if } \widetilde{u} \in \overline{V}, \widetilde{w} \in \widehat{V} \\ 1 - \alpha_{p(u)} & \text{if } \widetilde{u} \in \widehat{V}, \widetilde{w} \in \overline{V}, u = w \\ \alpha_{p(u)} & \text{if } \widetilde{u} \in \widehat{V}, p(u) \text{ is even}, \widetilde{w} = v_{win} \\ \alpha_{p(u)} & \text{if } \widetilde{u} \in \widehat{V}, p(u) \text{ is odd}, \widetilde{w} = v_{lose} \\ 1 & \text{if } \widetilde{u} = \widetilde{w} = v_{win} \text{ or } \widetilde{u} = \widetilde{w} = v_{lose} \\ 0 & \text{otherwise} \end{cases}$$

When the context is clear, we also address the SPG $(G, PA(p))$ and the SSG $(\widetilde{G}, RE(v_{win}))$ with G and $\widetilde{G}$ respectively.

Since all new vertices $\widehat{V} \uplus \{v_{win}, v_{lose}\}$ are random vertices, a strategy of either player in SPG G is a strategy in SSG $\widetilde{G}$ and vice versa. That is, there is a one-to-one relationship between strategies in G and $\widetilde{G}$. Hence, to keep notations simpler, we do not distinguish between strategies in G and $\widetilde{G}$.

A pair of strategies $\sigma, \gamma \in \Sigma_\exists \times \Sigma_\forall$ for Eve and Adam in G induces the sub-arena $G_{\sigma,\gamma}$. Similarly, we obtain $\widetilde{G}_{\sigma,\gamma}$. If U is an even or odd BSCC in SPG $G_{\sigma,\gamma}$, we denote with $\widetilde{U}$ what we call the associated *even pBSCC* or *odd pBSCC* in SSG $\widetilde{G}_{\sigma,\gamma}$ respectively. While those are not BSCCs, they correspond to the BSCC of the associated parity game, and we never consider the only true BSCCs of $\widetilde{G}_{\sigma,\gamma}$, i.e. $\{v_{win}\}$ and $\{v_{lose}\}$.

We continue with the example in Figure 2 and Figure 3. For the vertices v_0, v_1, and v_2, we assign priority 0, and for each remaining vertex v_i, $i \in \{3,4,5\}$, we assign priority i. An illustration of the corresponding sub-arena $\widetilde{G}_{\sigma,\gamma}$, induced by our gadget construction, is provided in Figure 5. Since we do not need to distinguish between different types of vertices, we use circles for all vertices. In $G_{\sigma,\gamma}$, the set $\{v_3, v_4, v_5\}$ forms an odd BSCC. We refer to the set $\{\widehat{v}_3, \overline{v}_3, \widehat{v}_4, \overline{v}_4, \widehat{v}_5, \overline{v}_5\}$ in $\widetilde{G}_{\sigma,\gamma}$ as the associated odd pBSCC with $\widehat{v}_3$, $\widehat{v}_4$, and $\widehat{v}_5$ having outgoing transitions to either v_{win} or v_{lose}.

3.2 Before Entering a pBSCC in SSG $\widetilde{G}$

Recall that when Eve and Adam follow pure memoryless strategies $\sigma \in \Sigma_\exists$ and $\gamma \in \Sigma_\forall$, the resulting sub-arenas $G_{\sigma,\gamma}$ and $\widetilde{G}_{\sigma,\gamma}$ can be viewed as finite Markov chains. We first focus on what happens before a play reaches a pBSCC in $\widetilde{G}_{\sigma,\gamma}$. Specifically, we give a lower bound on the probability of reaching an entry state of a pBSCC without entering a winning or losing sink. Later, in Lemma 11, we use this bound to determine a suitable value for α_0.

Intuitively, we show that the probability of entry is minimized in a classical worst-case scenario extending the sub-arena $\{v_0, v_1, v_2\}$ in Figure 3, and $\{\widehat{v}_0, \overline{v}_0, \widehat{v}_1, \overline{v}_1, \widehat{v}_2, \overline{v}_2\}$ in Figure 5. More precisely, we consider an original sub-arena $K_{\sigma,\gamma}$, depicted in Figure 6a with n states

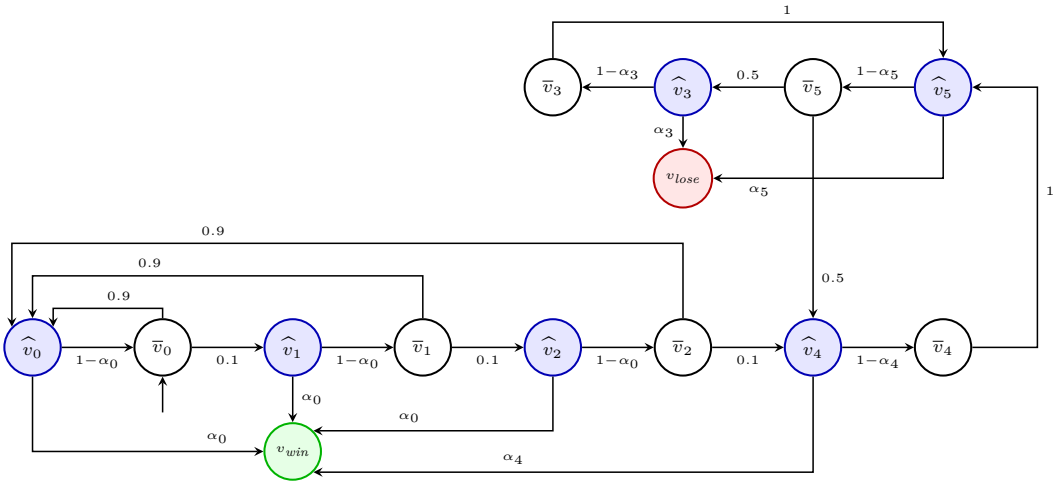


Figure 5 The sub-arena $\widetilde{G}_{\sigma,\gamma}$ induced by the gadget.

arranged in a sequence (as $\{v_0, v_1, v_2\}$ in Figure 3) before reaching a BSCC. Each state has the minimal probability δ_{min} of progressing to the next state and a maximal probability $1 - \delta_{\min}$ of returning to the initial state. All these states have parity value 0. Upon applying our gadget construction to obtain $\widetilde{K}_{\sigma,\gamma}$ in Figure 6b, we introduce random states ($\{\widehat{v}_0, \widehat{v}_1, \widehat{v}_2\}$ in Figure 5) that have the highest probability α_0 of going to the winning sink v_{win}.

In the following, let $(G, PA(p))$ be an SPG, and $(\widetilde{G}, RE(v_{win}))$ be its associated SSG. For all strategy pairs $\sigma, \gamma \in \Sigma_\exists \times \Sigma_\forall$, let $\widetilde{\Pr}^{\overline{v}}_{\sigma,\gamma}(crossPath)$ denote the probability for a play starting from $\overline{v} \in \overline{V}$ to reach a pBSCC in $\widetilde{G}$. Note that we never consider $\widehat{v} \in \widehat{V}$ as the starting vertex.

▶ **Lemma 6.** *For all strategy pairs $\sigma, \gamma \in \Sigma_\exists \times \Sigma_\forall$, for all $v \in V$, it holds:*

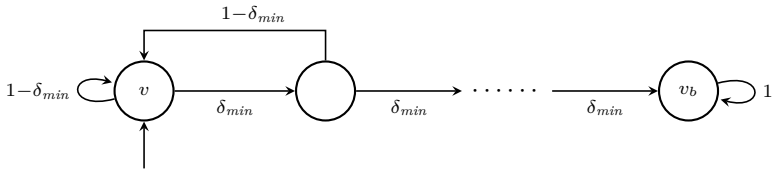
$$\widetilde{\Pr}^{\overline{v}}_{\sigma,\gamma}(crossPath) \geq \frac{(1 - x_0)x_0^n}{(1 - x_0) - (1 - x_0^n)x_1}$$

where $n = |V|$, $x_0 = \delta_{min}(1 - \alpha_0)$, $x_1 = (1 - \delta_{min})(1 - \alpha_0)$, and $\delta_{min} = \min_{u,v \in V} \{\Delta(u, v) \mid \Delta(u, v) > 0\}$.

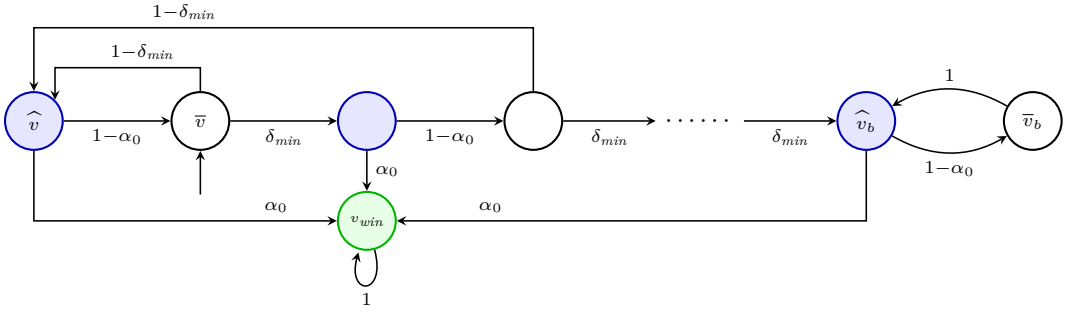
Sketch of Proof. We fix an arbitrary strategy pair $\sigma, \gamma \in \Sigma_\exists \times \Sigma_\forall$, and analyze the corresponding MC $\widetilde{G}_{\sigma,\gamma}$. We simplify the MC while either preserving or under-approximating the probability of reaching a pBSCC in $\widetilde{G}_{\sigma,\gamma}$. These steps merge all pBSCCs into a sink v_b, eliminate auxiliary states to simplify the MC, increase all values of α, and restructure transitions so that only one designated vertex can reach the sink v_b directly. We denote the resulting MC with $\widetilde{G}_4$. We then derive a lower bound on the probability of reaching v_b in $\widetilde{G}_4$, which provides a reachability lower bound in a template MC with absorbing sinks and bounded transition probabilities. As the reachability probabilities of v_b in $\widetilde{G}_4$ underapproximate those in $\widetilde{G}_{\sigma,\gamma}$, this yields the desired lower bound on $\widetilde{\Pr}^{\overline{v}}_{\sigma,\gamma}(crossPath)$.

3.3 Inside a pBSCC in SSG $\widetilde{G}$

We now focus on what happens after a play reaches a pBSCC in sub-arena $\widetilde{G}_{\sigma,\gamma}$. Specifically, we consider MCs and give a lower bound on the probability of reaching the winning sink after reaching an even pBSCC, and dually an upper bound on the probability of reaching the winning sink after reaching an odd pBSCC.



(a) The original sub-arena $K_{\sigma,\gamma}$.



(b) The sub-arena $\widetilde{K}_{\sigma,\gamma}$ induced by the gadget, where $\widetilde{\Pr}^{\overline{v}}_{\sigma,\gamma}(\mathit{crossPath})$ is minimized.

Figure 6 Example sub-arenas where the probability of *crossPath* is minimized.

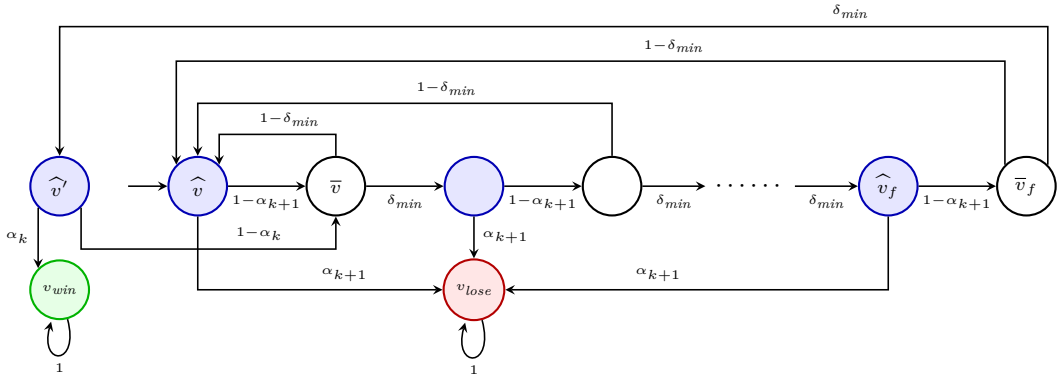


Figure 7 The MC where the lower bound on minimum *winEven* probability is attained.

The lower bound is attained in the MC shown in Figure 7, where k is an even parity value. There are $2n+1$ states in a line, and winning and losing sinks. Each white state has maximal probability $1-\delta_{\min}$ to return to the initial state, and otherwise proceeds to the next blue state. Each blue state, except $\widehat{v}'$, can with probability α_{k+1} go to the losing sink, and otherwise proceeds to the next white state. The special state $\widehat{v}'$ goes with probability α_k to v_{win}, and otherwise proceeds to $\overline{v}$. Unlike the case with $\{\widehat{v}_0, \overline{v}_0, \widehat{v}_1, \overline{v}_1, \widehat{v}_2, \overline{v}_2\}$ in Figure 5, this MC cannot be obtained by applying our gadget on some sub-arena $G_{\sigma,\gamma}$, and hence this bound is not guaranteed to be tight. More precisely, the outgoing transitions of $\widehat{v}$ indicate that v has an odd parity value, while $\widehat{v}'$ suggests otherwise. The upper bound is obtained by considering the same MC, where k is an odd parity value. Later, in Lemma 11, we use these two bounds to find suitable values for all α_k with $k \in \mathbb{N}$.

Let U be an even BSCC with smallest priority k in $G_{\sigma,\gamma}$ and $\widetilde{U}$ its associated pBSCC in $\widetilde{G}_{\sigma,\gamma}$. Let $\widetilde{\Pr}^{k,\widetilde{U}}_{\sigma,\gamma}(\mathit{winEven})$ denote the minimum probability of reaching the winning sink after reaching $\widetilde{U}$. That is, $\widetilde{\Pr}^{k,\widetilde{U}}_{\sigma,\gamma}(\mathit{winEven}) = \min\{\widetilde{\Pr}^{\widetilde{v}}_{\sigma,\gamma}(\mathit{Reach}(v_{win})) \mid \widetilde{v} \in \widetilde{U}\}$. We denote

it $\widetilde{\Pr}^{k,\widetilde{U}}_{\sigma,\gamma}(\mathit{winEven})$ when $\widetilde{U}$ is clear from context. Analogously, given an odd BSCC U with smallest priority k in $G_{\sigma,\gamma}$, we use $\widetilde{\Pr}^{k}_{\sigma,\gamma}(\mathit{winOdd})$ to denote the maximum probability of reaching the winning sink after reaching $\widetilde{U}$.

▶ **Lemma 7.** *For all strategy pairs $\sigma, \gamma \in \Sigma_\exists \times \Sigma_\forall$, for all even k, it holds:*

$$\widetilde{\Pr}^{k}_{\sigma,\gamma}(\mathit{winEven}) \geq (1-\alpha_{k+1}) \cdot \frac{(1-x_2)\cdot x_2^{n-1}\cdot x_4}{1-(x_2+x_3)+x_5\cdot x_2^n + t\cdot x_2^{n-1} - x_5\cdot x_2^{n-1}}$$

and for all odd k, it holds:

$$\widetilde{\Pr}^{k}_{\sigma,\gamma}(\mathit{winOdd}) \leq 1-(1-\alpha_{k+1}) \cdot \frac{(1-x_2)\cdot x_2^{n-1}\cdot x_4}{1-(x_2+x_3)+x_5\cdot x_2^n + t\cdot x_2^{n-1} - x_5\cdot x_2^{n-1}}$$

where $n = |V|$, $x_2 = \delta_{min}(1-\alpha_{k+1})$, $x_3 = (1-\delta_{min})(1-\alpha_{k+1})$, $x_4 = \delta_{min}\alpha_k$, $x_5 = \delta_{min}(1-\alpha_k) + (1-\delta_{min})(1-\alpha_{k+1})$, and δ_{min} is as before.

Sketch of Proof. The proof follows the same structure as the one for Lemma 6, applied to a BSCC in $\mathcal{M}_{\sigma,\gamma}$. We apply a similar four-step transformation and obtain a simplified MC where we can directly derive an upper bound. The lower bound comes as the dual.

3.4 Range of Winning Probabilities in the SSG

We now relate the winning probabilities in the constructed SSG $\widetilde{G}$ to the original SPG G. Intuitively, with a fixed strategy pair $\sigma, \gamma \in \Sigma_\exists \times \Sigma_\forall$, the value $\widetilde{\mathbb{P}}_{\sigma,\gamma}$ of the SSG $\widetilde{G}$ falls into a range around the value $\mathbb{P}_{\sigma,\gamma}$ of the SPG G, and the range size depends on the probabilities *crossPath*, *winEven* and *winOdd*.

▶ **Lemma 8.** *Let $x, y \in (0,1)$ such that $\widetilde{\Pr}^{\overline{v}}_{\sigma,\gamma}(\mathit{crossPath}) > x$ for all even k, and $\widetilde{\Pr}^{k}_{\sigma,\gamma}(\mathit{winEven}) \geq y$, and for all odd k, $\widetilde{\Pr}^{k}_{\sigma,\gamma}(\mathit{winOdd}) \leq 1-y$, then it holds:*

$$y\cdot \mathbb{P}^v_{\sigma,\gamma} - y + x\cdot y \;\leq\; \widetilde{\mathbb{P}}^{\overline{v}}_{\sigma,\gamma} \;\leq\; \mathbb{P}^v_{\sigma,\gamma} + 1 - x\cdot y$$

4 Reducing SPGs to SSGs

We now present the direct reduction from SPGs to SSGs. Let $G = ((V,E),(V_\exists, V_\forall, V_R), \Delta)$ be a stochastic arena, $p : V \to \mathbb{N}$ be a priority function, and $(G, PA(p))$ be an SPG. We construct the SSG $(\widetilde{G}, RE(v_{win}))$ using the gadget presented in Section 3.1. Section 4.1 presents a lower bound on the difference between winning probabilities associated to different strategy pairs in G. Section 4.2 presents the main theorem establishing the reduction, while Section 4.3 gives complexity bounds.

4.1 A Lower Bound on Different Strategies

We consider two strategy pairs $(\sigma,\gamma),(\sigma',\gamma') \in \Sigma_\exists \times \Sigma_\forall$ and show a general result on all such pair: if they yield different values in G, then there exists a lower bound on the difference between these values.

In the following, we assume for all $u \in V_R, v \in V$ that $\Delta(u,v)$ is a rational number $\frac{a_{u,v}}{b_{u,v}}$, where $a_{u,v} \in \mathbb{N}, b_{u,v} \in \mathbb{N}_+$ and $a_{u,v} \leq b_{u,v}$. Let $M = \max_{(u,v)\in E}\{b_{u,v}\}$ and $n = |V|$.

▶ **Lemma 9.** *For all* $(\sigma,\gamma),(\sigma',\gamma') \in \Sigma_\exists \times \Sigma_\forall$*, for all* $v \in V$*, the following holds:*

$$\mathbb{P}^v_{\sigma,\gamma} \neq \mathbb{P}^v_{\sigma',\gamma'} \Rightarrow |\mathbb{P}^v_{\sigma,\gamma} - \mathbb{P}^v_{\sigma',\gamma'}| > \frac{1}{(n!)^2 M^{2n^2}} = \epsilon$$

Proof. Let $\Pr^v_{\sigma,\gamma}(\mathit{enterEven})$ be the probability for a play starting from $v \in V$ to reach an even BSCC. It follows from Corollary 5 that for all $\sigma,\gamma \in \Sigma_\exists \times \Sigma_\forall$ and $v \in V$, we have $\mathbb{P}^v_{\sigma,\gamma} = \Pr^v_{\sigma,\gamma}(\mathit{enterEven})$. We can obtain $\Pr^v_{\sigma,\gamma}(\mathit{enterEven})$ by setting all vertices belonging to at least one even BSCC as the target set, and calculating the reachability probability. Calculating $\mathbb{P}^v_{\sigma,\gamma}$ is thus reduced to solving a linear equation system $x = Ax + b$ according to Theorem 1. We omit the details of A and b. Every non-zero entry of A and b is either 1, or $\frac{a_{u,v}}{b_{u,v}}$ for some $u,v \in V_R \times V$.

We use the following notations:

- Let $s = |b|$. It follows that $s < n$ since there is at least one vertex in a BSCC.
- Let $Q = I - A$. For $i \in 1,2,\ldots,n$ we denote the i-th row of Q with $Q[i]$, and the entry of Q at i-th row and j-th column with $Q[i,j]$. It can be written as $Q[i,j] = \frac{c_{i,j}}{d_{i,j}}$, where $|c_{i,j}|$ and $|d_{i,j}|$ are natural numbers bounded by M with $|c_{i,j}| \leq |d_{i,j}|$.
- We denote the i-th entry of b with $b[i]$. It can be written as $b[i] = \frac{c_{i,s+1}}{d_{i,s+1}}$, where $|c_{i,s+1}|$ and $|d_{i,s+1}|$ are natural numbers bounded by M with $|c_{i,s+1}| \leq |d_{i,s+1}|$.

The equation system can be written as:

$$Qx = b$$

We take an arbitrary row i, and write the i-th equation $Q[i] \cdot x = b[i]$ as follows:

$$\begin{bmatrix} \frac{c_{i,1}}{d_{i,1}} & \frac{c_{i,2}}{d_{i,2}} & \cdots & \frac{c_{i,s}}{d_{i,s}} \end{bmatrix} \cdot x = \frac{c_{i,s+1}}{d_{i,s+1}} \tag{1}$$

We multiply equation (1) with $\prod_{t=1}^{s+1} d_{i,t}$ to obtain:

- For all $j = 1,\ldots,s$, $Q[i,j]$ equals $(\prod_{t=1}^{s+1} d_{i,t})\frac{c_{i,j}}{d_{i,j}}$, an integer with absolute value bounded by M^{s+1}.
- For all $i = 1,\ldots,s$, $b[i]$ equals $(\prod_{t=1}^{s} d_{i,t})c_{i,s+1}$, an integer with absolute value bounded by M^{s+1}.

We apply this transformation to each row of the equation system, and write the new equation system as:

$$Q'x = b'$$

By Cramer's rule, for all $i = 1,2,\ldots,s$, we obtain:

$$x[i] = \frac{det(Q'_i)}{det(Q')}$$

where Q'_i is the matrix obtained by replacing the i-th column of Q' with the column vector b'. It follows that all entries of Q'_i are also integers with absolute values bounded by M^{s+1}.

Since $x[i]$ is a reachability probability, we have $x[i] \leq 1$. Following from the calculation of determinants, we obtain the following:

$$|det(Q'_i)| \leq |det(Q')| \leq s!(M^{s+1})^s < n!M^{n^2}$$

Therefore, if the equation system resulting from σ',γ' yields $x'[i] > x[i]$, we have:

$$x'[i] - x[i] > \frac{1}{(n!M^{n^2})^2} = \frac{1}{(n!)^2 M^{2n^2}}$$ ◀

4.2 Direct Reduction

We now establish the direct reduction from SPGs to SSGs.

▶ **Theorem 10** (Reducing SPGs to SSGs). *If for all $(\sigma, \gamma) \in \Sigma_\exists \times \Sigma_\forall$, and $v \in V$, the following conditions hold:*
1. $\widetilde{\Pr}^{v}_{\sigma,\gamma}(crossPath) > \frac{4-\epsilon}{4}$
2. $\widetilde{\Pr}^{k}_{\sigma,\gamma}(winEven) \geq \frac{4}{4+\epsilon}$ *for all even k, and* $\widetilde{\Pr}^{k}_{\sigma,\gamma}(winOdd) \leq 1 - \frac{4}{4+\epsilon}$ *for all odd k*

where $\epsilon = \frac{1}{(n!)^2 M^{2n^2}}$, then every optimal strategy $\sigma \in \Sigma_\exists$ of Eve in the SSG $(\widetilde{G}, RE(v_{win}))$ is also optimal in the SPG $(G, PA(p))$. The same holds for Adam.

Proof. We assume conditions 1 and 2 hold. We show that every optimal strategy $\sigma \in \Sigma_\exists$ in SSG $\widetilde{G}$ is also optimal in SPG G. We prove this by contraposition.

We take $\Sigma^*_\exists \subseteq \Sigma_\exists$ and $\Sigma^*_\forall \subseteq \Sigma_\forall$ as the sets of optimal strategies of Eve and Adam in $\widetilde{G}$. We obtain by Lemma 8 that for all $v \in V$ and all $\sigma, \gamma \in \Sigma^*_\exists \times \Sigma^*_\forall$, the following holds:

$$y \cdot \mathbb{P}^{v}_{\sigma,\gamma} - y + x \cdot y \;\leq\; \widetilde{\mathbb{P}}^{\overline{v}}_{\sigma,\gamma} \;\leq\; \mathbb{P}^{v}_{\sigma,\gamma} + 1 - x \cdot y$$

Since conditions 1 and 2 hold, we substitute x and y to obtain:

$$\widetilde{\mathbb{P}}^{\overline{v}}_{\sigma,\gamma} \leq \mathbb{P}^{v}_{\sigma,\gamma} + \frac{2\epsilon}{4+\epsilon} \tag{2}$$

If σ is not optimal in G, then there exists another strategy $\sigma' \in \Sigma_\exists$ and a vertex $v \in V$ such that $\mathbb{P}^{v}_{\sigma',\gamma} > \mathbb{P}^{v}_{\sigma,\gamma}$. It follows again from Lemma 8 that:

$$y\mathbb{P}^{v}_{\sigma',\gamma} - y + x \cdot y \;\leq\; \widetilde{\mathbb{P}}^{\overline{v}}_{\sigma',\gamma} \;\leq\; \mathbb{P}^{v}_{\sigma',\gamma} + 1 - x \cdot y \tag{3}$$

Furthermore, Lemma 9 yields:

$$\mathbb{P}^{v}_{\sigma',\gamma} > \mathbb{P}^{v}_{\sigma,\gamma} + \epsilon \tag{4}$$

As a result, we obtain the following:

$$\begin{aligned}
\widetilde{\mathbb{P}}^{\overline{v}}_{\sigma',\gamma} &\geq y \cdot \mathbb{P}^{v}_{\sigma',\gamma} - y + x \cdot y && \text{by (3)}\\
&> y \cdot (\mathbb{P}^{v}_{\sigma,\gamma} + \epsilon) - y + x \cdot y && \text{by (4)}\\
&= \frac{4}{4+\epsilon}(\mathbb{P}^{v}_{\sigma,\gamma} + \epsilon) - \frac{4}{4+\epsilon} \cdot \frac{\epsilon}{4}\\
&= \mathbb{P}^{v}_{\sigma,\gamma} - \frac{\epsilon}{4+\epsilon}\mathbb{P}^{v}_{\sigma,\gamma} + \frac{3\epsilon}{4+\epsilon}\\
&\geq \mathbb{P}^{v}_{\sigma,\gamma} + \frac{2\epsilon}{4+\epsilon}\\
&\geq \widetilde{\mathbb{P}}^{\overline{v}}_{\sigma,\gamma} && \text{by (2)}
\end{aligned}$$

It indicates that $\widetilde{\mathbb{P}}^{\overline{v}}_{\sigma',\gamma} > \widetilde{\mathbb{P}}^{\overline{v}}_{\sigma,\gamma}$, which contradicts the assumption that $\sigma \in \Sigma^*_\exists$. ◀

Until now, we have used the function α in Theorem 10, obtaining inequalities relating parity values in SPG G to transition probabilities in SSG $\widetilde{G}$. We now give requirements for α that satisfy all these inequalities.

▶ **Lemma 11.** *When the values of α are arranged as follows, the conditions in Theorem 10 are satisfied:*
1. *If $\alpha_0 \leq \frac{\delta^n_{min}}{8(n!)^2 M^{2n^2}}$, then condition 1 is satisfied.*
2. *If for all $k \in \mathbb{N}$, the following holds, then condition 2 is satisfied:*

$$\frac{\alpha_{k+1}}{\alpha_k} \leq \frac{\delta^n_{min}(1-\delta_{min})}{8(n!)^2 M^{2n^2}+1}.$$

Sketch of Proof. Both cases follow a similar structure. For α_0 (respectively α_{k+1}/α_k), we derive from the bound given by Lemma 6 (resp. Lemma 7) a corollary giving a bound that explicitly makes use of α. We then directly obtain the two cases of Lemma 11 from these two bounds. The full proof of this lemma, detailing how to compute function α can be found in Appendix A.1.

4.3 Complexity Considerations

To introduce complexity results, we first define size of a stochastic game G as $|G| = |V| + |E| + |\Delta|$ where $|\Delta|$ is the space needed to store the transition function (which may be stored in unary or binary). A now longstanding result shows that most stochastic game settings are polynomially reducible one to the other. In particular:

▶ **Theorem 12** (From Theorem 1 in [2]). *Solving stochastic parity games and solving simple stochastic games is polynomial-time equivalent. Either can be using unary or binary encoding.*

We show that the reduction we have introduced in this paper is polynomial with binary encoding. We recall that $M = \max_{(u,v)\in E}\{b_{u,v}\}$ and $n = |V|$.

▶ **Theorem 13.** *Given an SPG G, there exist polynomial values for function α that satisfy Theorem 10, such that the SSG $\widetilde{G}$ is of size $\mathcal{O}(n^5 \log M)$ in binary.*

Proof. Since $\delta_{min} \geq \frac{1}{M}$ and $1 - \delta_{min} \geq \frac{1}{2}$, the following is a valid instance of α, polynomial in G (and polynomial in the transition probabilities appearing in G) under binary encoding:

$$\forall k \in \mathbb{N},\ \alpha_k = \left(\frac{1}{16(n!)^2 M^{2n^2+n} + 1}\right)^{k+1}$$

Then the size of the SSG $\widetilde{G}$ is:

$$\begin{aligned}|\widetilde{G}| &= |\widetilde{V} \uplus \{v_{win}, v_{lose}\}| + |\widetilde{E}| + |\widetilde{\Delta}| \\ &= \mathcal{O}(n) + \mathcal{O}(n^2) + \mathcal{O}(n^2) \cdot \mathcal{O}(n \cdot (n \log n + n^2 \log M)) \\ &= \mathcal{O}(n^5 \log M)\end{aligned}$$

◀

According to [2], quantitative SSGs under unary and binary encoding are in the same complexity class, and so in $\mathbf{NP} \cap \mathbf{coNP}$ [18]. We thus obtain that our reduction yields an $\mathbf{NP} \cap \mathbf{coNP}$ algorithm for solving SPGs.

5 Epilogue

We have given a polynomial reduction from quantitative SPGs to quantitative SSGs, taking inspiration from a gadget used in [9] to obtain a reduction from deterministic PGs to quantitative SSGs. After fixing a pair of strategies, the values of both the SPG and the SSG are determined, but the construction of the SSG makes it difficult to establish coinciding values. Using these fixed strategies, we showed that the value of the SSG falls into a range around the value of the SPG, where this range depends on the probability to reach a pBSCC of the SSG and the minimum probability to reach a winning sink in pBSCCs of the SSG. When considering all possible strategy pairs, we obtained a lower bound ϵ on their value differences in the SPG, by restricting transition probabilities to rational numbers and analyzing reachability equation systems of Markov chains. We then showed that by arranging transition probabilities of the SSG properly in terms of the size of the SPG, its smallest probability, and ϵ, the value ranges of different strategy pairs can be narrowed so that they do not overlap. In this case, a reduction from SPGs to SSGs is achieved.

Although under unary encoding, exponential numbers can be introduced into the probability function α of the newly constructed SSGs, both reductions are polynomial. Hence, our construction yields an $\mathbf{NP} \cap \mathbf{coNP}$ algorithm in both qualitative and quantitative SSGs under unary and binary encoding, substantiating the complexity results from previous works [18, 2, 9]. .

Our result enables solving SPGs by first reducing them to SSGs and then applying algorithms for SSGs. However, its implementability is in question, due to the possibly huge representation of α. Our reduction also captures the transformation from an MDP with a parity objective into an SSG. As we assume the minimum transition probability to be $\delta_{min} \in (0, \frac{1}{2}]$ in the original SPG, we cannot capture the subcase of reducing DPGs to quantitative SSGs. Although our reduction is unlikely to be leveraged to effectively solve SPGs in practice, some improvements are possible. First, we have not formally examined the optimal arrangement of α. It is possible to find the weakest requirements on α so that the reductions are correct, thus optimizing possible implementations. Second, some specific cases lead to very small values of α. These cases are similar to the ones that can challenge classical MDP solvers using VI, and so we can benefit from any family of arena structure where these cases are avoided, leading to implementable valuations of α.

References

1 Rabah Amir. Stochastic games in economics and related fields: an overview. *Stochastic Games and Applications*, pages 455–470, 2003.

2 Daniel Andersson and Peter Bro Miltersen. The complexity of solving stochastic games on graphs. In *International Symposium on Algorithms and Computation*, pages 112–121. Springer, 2009. doi:10.1007/978-3-642-10631-6_13.

3 Muqsit Azeem, Alexandros Evangelidis, Jan Křetínskỳ, Alexander Slivinskiy, and Maximilian Weininger. Optimistic and topological value iteration for simple stochastic games. In *International Symposium on Automated Technology for Verification and Analysis*, pages 285–302. Springer, 2022.

4 Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT press, 2008.

5 Christel Baier, Joachim Klein, Linda Leuschner, David Parker, and Sascha Wunderlich. Ensuring the reliability of your model checker: Interval iteration for Markov decision processes. In *International Conference on Computer Aided Verification*, volume 10426, pages 160–180. Springer, 2017. doi:10.1007/978-3-319-63387-9_8.

6 Raphaël Berthon, Joost-Pieter Katoen, and Zihan Zhou. A direct reduction from stochastic parity games to simple stochastic games, 2025. arXiv:2506.06223.

7 Patricia Bouyer, Youssouf Oualhadj, Mickael Randour, and Pierre Vandenhove. Arena-independent finite-memory determinacy in stochastic games. *Log. Methods Comput. Sci.*, 19(4), 2023. doi:10.46298/LMCS-19(4:18)2023.

8 Cristian S Calude, Sanjay Jain, Bakhadyr Khoussainov, Wei Li, and Frank Stephan. Deciding parity games in quasipolynomial time. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, pages 252–263, 2017. doi:10.1145/3055399.3055409.

9 Krishnendu Chatterjee and Nathanaël Fijalkow. A reduction from parity games to simple stochastic games. *Electronic Proceedings in Theoretical Computer Science*, 54:74–86, 2011. doi:10.4204/eptcs.54.6.

10 Krishnendu Chatterjee and Thomas A Henzinger. Strategy improvement and randomized subexponential algorithms for stochastic parity games. In *Annual Symposium on Theoretical Aspects of Computer Science*, pages 512–523. Springer, 2006. doi:10.1007/11672142_42.

11 Krishnendu Chatterjee and Thomas A Henzinger. Reduction of stochastic parity to stochastic mean-payoff games. *Information Processing Letters*, 106(1):1–7, 2008. doi:10.1016/J.IPL.2007.08.035.

12 Krishnendu Chatterjee, Thomas A Henzinger, Barbara Jobstmann, and Arjun Radhakrishna. Gist: A solver for probabilistic games. In *Computer Aided Verification: 22nd International Conference*, volume 6174, pages 665–669. Springer, 2010. doi:10.1007/978-3-642-14295-6_57.

13 Krishnendu Chatterjee, Marcin Jurdziński, and Thomas A. Henzinger. Simple stochastic parity games. In Matthias Baaz and Johann A. Makowsky, editors, *Computer Science Logic*, volume 2803, pages 100–113, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. doi:10.1007/978-3-540-45220-1_11.

14 Krishnendu Chatterjee, Marcin Jurdzinski, and Thomas A Henzinger. Quantitative stochastic parity games. In *SODA*, volume 4, pages 121–130, 2004. URL: http://dl.acm.org/citation.cfm?id=982792.982808.

15 Krishnendu Chatterjee, Rupak Majumdar, and Marcin Jurdzinski. On Nash equilibria in stochastic games. In *Computer Science Logic*, volume 3210 of *Lecture Notes in Computer Science*, pages 26–40. Springer, 2004. doi:10.1007/978-3-540-30124-0_6.

16 Krishnendu Chatterjee and Nir Piterman. Combinations of qualitative winning for stochastic parity games. In *30th International Conference on Concurrency Theory*, volume 140 of *LIPIcs*, pages 6:1–6:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPICS.CONCUR.2019.6.

17 Anne Condon. On algorithms for simple stochastic games. *Advances in Computational Complexity Theory*, 13:51–72, 1990. doi:10.1090/DIMACS/013/04.

18 Anne Condon. The complexity of stochastic games. *Information and Computation*, 96(2):203–224, 1992. doi:10.1016/0890-5401(92)90048-K.

19 Wojciech Czerwiński, Laure Daviaud, Nathanaël Fijalkow, Marcin Jurdziński, Ranko Lazić, and Paweł Parys. Universal trees grow inside separating automata: Quasi-polynomial lower bounds for parity games. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2333–2349. SIAM, 2019.

20 Matthew Darlington, Kevin D. Glazebrook, David S. Leslie, Rob Shone, and Roberto Szechtman. A stochastic game framework for patrolling a border. *Eur. J. Oper. Res.*, 311(3):1146–1158, 2023. doi:10.1016/J.EJOR.2023.06.011.

21 Julia Eisentraut, Edon Kelmendi, Jan Křetínskỳ, and Maximilian Weininger. Value iteration for simple stochastic games: Stopping criterion and learning algorithm. *Information and Computation*, 285:104886, 2022. doi:10.1016/J.IC.2022.104886.

22 Kousha Etessami, Marta Z. Kwiatkowska, Moshe Y. Vardi, and Mihalis Yannakakis. Multi-objective model checking of Markov decision processes. *Log. Methods Comput. Sci.*, 4(4), 2008. doi:10.2168/LMCS-4(4:8)2008.

23 Ernst Moritz Hahn, Mateo Perez, Sven Schewe, Fabio Somenzi, Ashutosh Trivedi, and Dominik Wojtczak. Model-free reinforcement learning for stochastic parity games. In *31st International Conference on Concurrency Theory*, volume 171 of *LIPIcs*, pages 21:1–21:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPICS.CONCUR.2020.21.

24 Arnd Hartmanns and Benjamin Lucien Kaminski. Optimistic value iteration. In *Computer Aided Verification - 32nd International Conference*, volume 12225 of *Lecture Notes in Computer Science*, pages 488–511. Springer, 2020. doi:10.1007/978-3-030-53291-8_26.

25 Christian Hensel, Sebastian Junges, Joost-Pieter Katoen, Tim Quatmann, and Matthias Volk. The probabilistic model checker storm. *International Journal on Software Tools for Technology Transfer*, pages 1–22, 2021.

26 Marcin Jurdziński. Deciding the winner in parity games is in **UP** ∩ **co-UP**. *Information Processing Letters*, 68(3):119–124, 1998.

27 Marcin Jurdziński and Ranko Lazić. Succinct progress measures for solving parity games. In *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–9. IEEE, 2017.

28 Mark Kattenbelt, Marta Kwiatkowska, Gethin Norman, and David Parker. A game-based abstraction-refinement framework for Markov decision processes. *Formal Methods in System Design*, 36:246–280, 2010. doi:10.1007/S10703-010-0097-6.

29 Jan Křetínskỳ, Emanuel Ramneantu, Alexander Slivinskiy, and Maximilian Weininger. Comparison of algorithms for simple stochastic games. *Information and Computation*, 289:104885, 2022. doi:10.1016/J.IC.2022.104885.

30 Marta Kwiatkowska, Gethin Norman, David Parker, and Gabriel Santos. PRISM-games 3.0: Stochastic game verification with concurrency, equilibria and time. In *Computer Aided Verification - 32nd International Conference*, volume 12225 of *Lecture Notes in Computer Science*, pages 475–487. Springer, 2020. doi:10.1007/978-3-030-53291-8_25.

31 Marta Z. Kwiatkowska, Gethin Norman, and David Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *Computer Aided Verification - 23rd International Conference*, volume 6806 of *Lecture Notes in Computer Science*, pages 585–591. Springer, 2011. doi:10.1007/978-3-642-22110-1_47.

32 Karoliina Lehtinen. A modal μ perspective on solving parity games in quasi-polynomial time. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 639–648, 2018. doi:10.1145/3209108.3209115.

33 David S Leslie, Steven Perkins, and Zibo Xu. Best-response dynamics in zero-sum stochastic games. *Journal of Economic Theory*, 189:105095, 2020. doi:10.1016/J.JET.2020.105095.

34 Rupak Majumdar, Kaushik Mallik, Anne-Kathrin Schmuck, and Sadegh Soudjani. Symbolic control for stochastic systems via finite parity games. *Nonlinear Analysis: Hybrid Systems*, 51:101430, 2024. doi:10.1016/j.nahs.2023.101430.

35 Donald A Martin. The determinacy of Blackwell games. *The Journal of Symbolic Logic*, 63(4):1565–1581, 1998. doi:10.2307/2586667.

36 Pawel Parys. Parity games: Zielonka's algorithm in quasi-polynomial time. In *44th International Symposium on Mathematical Foundations of Computer Science*, volume 138 of *LIPIcs*, pages 10:1–10:13. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPICS.MFCS.2019.10.

37 Kittiphon Phalakarn, Toru Takisaka, Thomas Haas, and Ichiro Hasuo. Widest paths and global propagation in bounded value iteration for stochastic games. In *Computer Aided Verification: 32nd International Conference*, pages 349–371. Springer, 2020. doi:10.1007/978-3-030-53291-8_19.

38 Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

39 Tim Quatmann and Joost-Pieter Katoen. Sound value iteration. In *Computer Aided Verification - 30th International Conference*, volume 10981 of *Lecture Notes in Computer Science*, pages 643–661. Springer, 2018. doi:10.1007/978-3-319-96145-3_37.

40 Lloyd S Shapley. Stochastic games. *Proceedings of the National Academy of Sciences*, 39(10):1095–1100, 1953.

41 Frédéric Simard, Josée Desharnais, and François Laviolette. General cops and robbers games with randomness. *Theor. Comput. Sci.*, 887:30–50, 2021. doi:10.1016/J.TCS.2021.06.043.

42 Michael Ummels and Dominik Wojtczak. The complexity of Nash equilibria in stochastic multiplayer games. *Log. Methods Comput. Sci.*, 7(3), 2011. doi:10.2168/LMCS-7(3:20)2011.

43 Uri Zwick and Mike Paterson. The complexity of mean payoff games on graphs. *Theoretical Computer Science*, 158(1-2):343–359, 1996. doi:10.1016/0304-3975(95)00188-3.

A Appendix

We present how we compute values for function α. We include the details, hoping these techniques may later be modified to yield better values on specific classes of games.

A.1 Proof of Lemma 11

▶ **Lemma 11.** *When the values of α are arranged as follows, the conditions in Theorem 10 are satisfied:*

1. *If $\alpha_0 \leq \frac{\delta_{min}^n}{8(n!)^2 M^{2n^2}}$, then condition 1 is satisfied.*
2. *If for all $k \in \mathbb{N}$, the following holds, then condition 2 is satisfied:*

$$\frac{\alpha_{k+1}}{\alpha_k} \leq \frac{\delta_{min}^n(1-\delta_{min})}{8(n!)^2 M^{2n^2}+1}.$$

A.1.1 Arranging α_0

We start by giving a bound on $\widetilde{\Pr}^{\overline{v}}_{\sigma,\gamma}(crossPath)$ that involves α_0. To do so, we make use of Lemma 6.

▶ **Corollary 14** (Another Lower Bound of *crossPath* Probability)**.** *For all strategy pairs $\sigma, \gamma \in \Sigma_\exists \times \Sigma_\forall$, for all $\overline{v} \in \overline{V}$, the following holds:*

$$\widetilde{\Pr}^{\overline{v}}_{\sigma,\gamma}(crossPath) > \frac{\delta_{min}^n(1-\alpha_0)^{n+1}}{2\alpha_0 + \delta_{min}^n(1-\alpha_0)^{n+1}}$$

Proof. We scale down $\widetilde{\Pr}^{\overline{v}}_{\sigma,\gamma}(crossPath)$ as follows:

$$\begin{aligned}
&\widetilde{\Pr}^{\overline{v}}_{\sigma,\gamma}(crossPath) \\
&\geq \frac{(1-x_0)x_0^n}{(1-x_0)-(1-x_0^n)x_1} && \text{Lemma 6} \\
&= \frac{(1-\delta_{min}(1-\alpha_0))\delta_{min}^n(1-\alpha_0)^n}{\alpha_0 + (1-\delta_{min})\delta_{min}^n(1-\alpha_0)^{n+1}} \\
&> \frac{(1-\delta_{min})(1-\alpha_0)\delta_{min}^n(1-\alpha_0)^n}{\alpha_0 + (1-\delta_{min})\delta_{min}^n(1-\alpha_0)^{n+1}} && \text{since } 1-\delta_{min}(1-\alpha_0) > (1-\delta_{min})(1-\alpha_0) \\
&= \frac{\delta_{min}^n(1-\alpha_0)^{n+1}}{\frac{\alpha_0}{1-\delta_{min}} + \delta_{min}^n(1-\alpha_0)^{n+1}} \\
&\geq \frac{\delta_{min}^n(1-\alpha_0)^{n+1}}{2\alpha_0 + \delta_{min}^n(1-\alpha_0)^{n+1}} && \text{since } 1-\delta_{min} \geq \tfrac{1}{2}
\end{aligned}$$

◀

We can now arrange α_0. It follows from Corollary 14 that:

$$\widetilde{\Pr}^{\overline{v}}_{\sigma,\gamma}(crossPath) > \frac{\delta_{min}^n(1-\alpha_0)^{n+1}}{2\alpha_0 + \delta_{min}^n(1-\alpha_0)^{n+1}}$$

Therefore to show:

$$\widetilde{\Pr}^{\overline{v}}_{\sigma,\gamma}(crossPath) > \frac{4-\epsilon}{4} \tag{5}$$

where $\epsilon = \frac{1}{(n!)^2 M^{2n^2}}$, it suffices to show that:

$$\frac{\delta_{min}^n(1-\alpha_0)^{n+1}}{2\alpha_0 + \delta_{min}^n(1-\alpha_0)^{n+1}} \geq \frac{4-\epsilon}{4} \tag{6}$$

which can be further simplified as:

$$\epsilon \geq \frac{8\alpha_0}{2\alpha_0 + \delta_{min}^n(1-\alpha_0)^{n+1}} \tag{7}$$

We show that when $\alpha_0 \leq \frac{\delta_{min}^n}{8(n!)^2 M^{2n^2}}$, inequalit (7) holds. We start with the right side:

$$
\begin{aligned}
&\frac{8\alpha_0}{2\alpha_0 + \delta_{min}^n(1-\alpha_0)^{n+1}} \\
\leq\ &\frac{8\alpha_0}{2\alpha_0 + \delta_{min}^n(1-(n+1)\alpha_0)} && \textit{follows from Bernoulli's inequality} \\
=\ &\frac{8\alpha_0}{\alpha_0 + \delta_{min}^n + \alpha_0(1-(n+1)\delta_{min}^n)} \\
\leq\ &\frac{8\alpha_0}{\alpha_0 + \delta_{min}^n} && \textit{since } 1-(n+1)\delta_{min}^n > 0 \\
\leq\ &\frac{\frac{\delta_{min}^n}{(n!)^2 M^{2n^2}}}{\frac{\delta_{min}^n}{8(n!)^2 M^{2n^2}} + \delta_{min}^n} \\
=\ &\frac{1}{\frac{1}{8} + (n!)^2 M^{2n^2}} \\
\leq\ &\frac{1}{(n!)^2 M^{2n^2}} = \epsilon
\end{aligned}
$$

Therefore we obtain that when $\alpha_0 \leq \frac{\delta_{min}^n}{8(n!)^2 M^{2n^2}}$, inequality (5) holds, and thus condition (1) is satisfied.

A.1.2 Arranging α_{k+1}/α_k

We start by getting a lower bound on $\widetilde{\Pr}^k_{\sigma,\gamma}(\mathit{winEven})$ for even k's, which makes use of α_{k+1} and α_k. The reasoning for odd k's is symmetric. To do so, we use Lemma 7.

▶ **Corollary 15** (Another Lower Bound of *winEven* Probability). *For all strategy pairs $\sigma, \gamma \in \Sigma \times \Gamma$, for all even k, the following holds:*

$$\widetilde{\Pr}^k_{\sigma,\gamma}(\mathit{winEven}) > \frac{\delta_{min}^n(1-\delta_{min}) - \frac{\alpha_{k+1}}{\alpha_k}}{\delta_{min}^n(1-\delta_{min}) + \frac{\alpha_{k+1}}{\alpha_k}}$$

Proof. We scale down the right side as follows:

$$
\begin{aligned}
&\widetilde{\Pr}^k_{\sigma,\gamma}(\mathit{winEven}) \\
\geq\ &(1-\alpha_{k+1}) \cdot \frac{(1-x_2)\cdot x_2^{n-1}\cdot x_4}{1-(x_2+x_3)+x_5\cdot x_2^n + t\cdot x_2^{n-1} - x_5\cdot x_2^{n-1}} && \textit{Lemma 7} \\
=\ &\frac{(1-\delta_{min}+\delta_{min}\alpha_{k+1})\cdot \delta_{min}^{n-1}(1-\alpha_{k+1})^n \cdot \delta_{min}\alpha_k}{\alpha_{k+1} + \delta_{min}^{n-1}(1-\alpha_{k+1})^{n-1}(\delta_{min}^2(-\alpha_k+\alpha_{k+1}+\alpha_k\alpha_{k+1}-\alpha_{k+1}^2) + \delta_{min}(\alpha_k - 2\alpha_{k+1}+\alpha_{k+1}^2))} \\
>\ &\frac{(1-\delta_{min})\cdot \delta_{min}^n \cdot (1-\alpha_{k+1})^{n+1}\cdot \alpha_k}{\alpha_{k+1} + \delta_{min}^n \cdot (\delta_{min}(-\alpha_k+\alpha_{k+1}+\alpha_k\alpha_{k+1}-\alpha_{k+1}^2) + (\alpha_k - 2\alpha_{k+1}+\alpha_{k+1}^2))} \\
& && \textit{since } 1-\delta_{min}+\delta_{min}\alpha_{k+1} > (1-\delta_{min})(1-\alpha_{k+1}) \textit{ and } (1-\alpha_{k+1})^{n-1} < 1 \\
\geq\ &\frac{(1-\delta_{min})\delta_{min}^n\alpha_k(1-(n+1)\alpha_{k+1})}{\alpha_{k+1} + \delta_{min}^n(\alpha_k(1-\delta_{min}) - \alpha_{k+1}(2-\alpha_{k+1}-\delta_{min}(1+\alpha_k-\alpha_{k+1})))} \\
& && \textit{follows from Bernoulli inequality} \\
>\ &\frac{(1-\delta_{min})\delta_{min}^n\alpha_k - (n+1)\delta_{min}^n(1-\delta_{min})\alpha_k\alpha_{k+1}}{\alpha_{k+1} + (1-\delta_{min})\delta_{min}^n\alpha_k} \\
& && \textit{since } \alpha_{k+1}(2-\alpha_{k+1}-\delta_{min}(1+\alpha_k-\alpha_{k+1})) > 0 \\
>\ &\frac{\delta_{min}^n\alpha_k(1-\delta_{min}) - \alpha_{k+1}}{\delta_{min}^n\alpha_k(1-\delta_{min}) + \alpha_{k+1}} && \textit{since } (n+1)\delta_{min}^n(1-\delta_{min})\alpha_k < 1 \\
=\ &\frac{\delta_{min}^n(1-\delta_{min}) - \frac{\alpha_{k+1}}{\alpha_k}}{\delta_{min}^n(1-\delta_{min}) + \frac{\alpha_{k+1}}{\alpha_k}}
\end{aligned}
$$

◀

We now arrange α_{k+1}/α_k. It follows from Corollary 15 that for all strategy pairs $\sigma, \gamma \in \Sigma \times \Gamma$, for all even k, the following holds:

$$\widetilde{\Pr}^k_{\sigma,\gamma}(winEven) > \frac{\delta^n_{min}(1-\delta_{min}) - \frac{\alpha_{k+1}}{\alpha_k}}{\delta^n_{min}(1-\delta_{min}) + \frac{\alpha_{k+1}}{\alpha_k}} \tag{8}$$

Therefore to show:

$$\widetilde{\Pr}^k_{\sigma,\gamma}(winEven) \geq \frac{4}{4+\epsilon} \tag{9}$$

it suffices to show that for all $k \in \mathbb{N}$:

$$\frac{\delta^n_{min}(1-\delta_{min}) - \frac{\alpha_{k+1}}{\alpha_k}}{\delta^n_{min}(1-\delta_{min}) + \frac{\alpha_{k+1}}{\alpha_k}} \geq \frac{4}{4+\epsilon} \tag{10}$$

We denote $\frac{\alpha_{k+1}}{\alpha_k}$ with r in the following calculation. Inequality 10 can be further simplified to:

$$\frac{\epsilon}{4+\epsilon} \geq \frac{2r}{(1-\delta_{min})\delta^n_{min} + r} \tag{11}$$

and can be finally simplified to:

$$r \leq \frac{(1-\delta_{min})\delta^n_{min}}{8(n!)^2 M^{2n^2} + 1} \tag{12}$$

Therefore we obtain that if for all $k \in \mathbb{N}$, the following holds:

$$\frac{\alpha_{k+1}}{\alpha_k} \leq \frac{(1-\delta_{min})\delta^n_{min}}{8(n!)^2 M^{2n^2} + 1}$$

then inequality (9) holds, and thus condition (2) is satisfied.

Abstract Subtyping for Asynchronous Multiparty Sessions

Laura Bocchi
University of Kent, Canterbury, UK

Andy King
University of Kent, Canterbury, UK

Maurizio Murgia
Gran Sasso Science Institute, L'Aquila, Italy

Simon Thompson
University of Kent, Canterbury, UK

Abstract

Session subtyping answers the question of whether a program in a communicating system can be safely substituted for another, when their communication behaviour is described by session types. Asynchronous session subtyping is undecidable, even for two participants, hence the interest in sound, but incomplete, subtyping algorithms. Asynchronous multiparty subtyping can be formulated by decomposing session types into single input and output types which preclude, respectively, external and internal choice. This paper shows how abstract interpretation can sit atop this approach and how it leads to an algorithm that can prove subtyping for intricate communication patterns.

2012 ACM Subject Classification Theory of computation → Program reasoning; Theory of computation → Process calculi

Keywords and phrases asynchrony, session subtyping, automata, abstract interpretation

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2025.10

Funding This work was supported by EPSRC project EP/T014512/1 (STARDUST) and the PRIN 2022 PNRR project DeLiCE (F53D23009130001).

1 Introduction

A significant challenge in message-passing concurrency is ensuring that each process in the system adheres to a desired protocol and is free from communication errors. Session types have proved to be powerful in addressing this challenge: they provide a theory for designing protocols with desirable behavioural guarantees, and are directly applicable to programming languages (notably Java [31], Python [22], Rust [33], Go [34] and Erlang [24, 37]) via static type-checking, runtime monitoring, or API generation. Session types were initially proposed for binary sessions involving only two roles (aka participants) [29], and were later extended to multiparty sessions, to allow for many roles [30]. Multiparty session types produce, via automated projection from a global protocol, a safe (by construction) realisation given as a *collection of local types*, one for each role. In this work we use this as a starting point: a collection of local types whose communications are *safe* in the sense that no role will ever: (1) get stuck (until it terminates); and (2) receive an unexpected message.

To illustrate protocol realisation, consider the three local types below: a server S, a client C, and a logger L, where s, c, and l denote the respective roles. The server can initially receive two kinds of messages from the client: $c?\mathrm{next}$ for more feeds or $c?\mathrm{stop}$ for termination. Feed requests are served with a feed ($c!\mathrm{feed}$) and waiting for further requests. The logging process gets regular updates by the server and sends a final summary to the client:

© Laura Bocchi, Andy King, Maurizio Murgia, and Simon Thompson;
licensed under Creative Commons License CC-BY 4.0

$$S = (c?\text{next}.\, l!\text{next}.\, c!\text{feed}.\, S \;+\; c?\text{stop}.\, l!\text{stop}) \qquad L = (s?\text{next}.\, L + s?\text{stop}.\, c!\text{sum})$$
$$C = (s!\text{next}.\, s?\text{feed}.\, C + s!\text{stop}.\, l?\text{sum})$$

We omit formalities on the semantics for the purposes of this introduction, and instead use color coding to hint at the correspondence between pairs of send/receive actions. This correspondence ensures that, in a system (S, C, L) behaving as the parallel composition of the three local types, no role ever gets stuck or receives an unexpected message.[1]

A fundamental problem in the application of session types is checking whether the implementation of one component in a distributed system can be substituted for another, without compromising safety of the overarching protocol. This problem can be addressed at the level of types and formulated as *session subtyping* [13, 21, 25, 26, 27]. Session subtyping establishes a preorder relation on local types: C' is a subtype of C, written $C' \leq C$, if a program with type C can be *safely* substituted by a program with type C'. Specifically, given a safe protocol (S, C, L), $C' \leq C$ ensures that (S, C', L) is also safe. For example, consider the variant $C' = s!\text{next}.\, (s?\text{feed}.\, s!\text{stop}.\, l?\text{sum} \;+\; s?\text{else})$ of C given earlier. With respect to C, C' has: (1) *less* send options (initially it can only select next, and after receiving one feed it can only select stop), (2) *more* receive options (the additional option $s?\text{else}$). This notion of substitutability is *covariant on send actions and contravariant on receive actions* [25]. In fact, one can verify $C' \leq C$, which ensures that if (S, C, L) is safe then so is (S, C', L).

Asynchronous session types are particularly interesting due to the use of FIFO channels in distributed systems and languages such as Go and Rust. When considering asynchronous session types, channels are unidirectional and unbounded (hence two are used for each pair of roles, one in each direction). For example, the asynchronous protocol (S, C', L) behaves as the parallel composition of 3 roles communicating over 6 unidirectional channels, where a send action enqueues a message to a channel and a receive action dequeues it.

Establishing whether subtyping holds is non-trivial, especially when communication is asynchronous. Recent work [27] has provided, for the first time, a *sound and complete* definition of Multiparty Asynchronous session Subtyping (MAS). This definition of session subtyping is sound and complete in the sense that it precisely capture only and all the safety-preserving modification one can perform on a local type. Besides co- and contra-variance, MAS enables some actions to be *swapped*, reflecting the asynchronous nature of interaction. MAS allows two kinds of swapping:

SW1 a role can anticipate an output action with respect to an input action;

SW2 a role can swap actions of the same directions from/to different roles.

Below, C'' is a variant of C' where the client *anticipates* the end-of-session request $s!\text{stop}$ before the processing of the feed $c?\text{next}$. This swap may be desirable to allow the client to process the feed from its own local queue at a later stage, and is allowed by SW1. S' is a variant of S where the send actions $c!\text{feed}$ and $l!\text{next}$ have been swapped, which is allowed by SW2. Indeed $C'' \leq C$ and $S' \leq S$.

$$C'' = (s!\text{next}.\, s!\text{stop}.\, s?\text{feed}.\, l?\text{sum}) \qquad S' = (c?\text{next}.\, c!\text{feed}.\, l!\text{next}.\, S' \;+\; c?\text{stop}.\, l!\text{stop})$$

This flexibility comes at a price: for interactions over unbounded channels MAS is *undecidable* [7, 35]. The subtyping problem can be viewed as a simulation game between the candidate subtype and the supertype, in which supertype is required to mirror any action

[1] Session types also accompany labels with payloads of type String, Int, higher-order, etc. We omit these details in our presentation as they are not relevant to subtyping.

performed by the candidate subtype. Consider checking whether C''' is a subtype of C where:

$$C''' = s!\text{next}.\, s!\text{next}.\, s?\text{feed}.\, C'''$$

The client requests feeds at double the pace of its own feed processing time. It is actually the case that $C''' \leq C$ according to asynchronous subtyping [7, 27]. However to verify $C''' \leq C$, one needs to show that C can follow the actions of C''', albeit with send actions being delayed. Due to unbounded channel size, an unbounded number of input actions (the feed processing) can be deferred, yielding an infinite space of possible interleavings.

Sound algorithms have emerged which aim to establish subtyping in practical scenarios. Most of these algorithms are limited to *binary* sessions (only two roles) and follow the definition of binary subtyping in [13]. The definition in [13] is *operational* in nature, based on simulation trees, which makes it relatively straightforward to translate it into a practical subtyping algorithm [3, 6]. Unlike the binary case [13], the multiparty definition of MAS in [27] is declarative. While MAS concisely expresses an intrinsically complex problem, it is also highly non-intuitive. Therefore, it is unrealistic to expect a developer to be able to check the correctness of their code substitutions using the definition directly (beside the fact that the problem is undecidable). MAS can only be practically used if a test for subtyping can be *automated*, and since the problem is undecidable, the solution will necessarily entail approximation. This paper has exactly this purpose: automation of MAS. Automation still raises the question "how can one trust the algorithm?" Our paper also answers this: our black-box (automatically) generates a *certificate* which can be, if needed, further checked off with a proof assistant [23].

In this introduction, to clarify the purpose of our work, we have provided an overview of the problem space, and hinted at the underlying asynchronous communication model and the notion of safety that subtyping preserves. However, these details are neither needed nor used in the technical development which follows, since our work builds entirely on the declarative definition of MAS [27] that elegantly abstracts over the communication model.

1.1 Challenges and contributions

The undecidability of MAS is unfortunate, as safety-preserving substitutions are most valuable in the asynchronous setting to check the validity of performance optimisations, namely sending messages earlier or postponing receives [38]. A MAS subtyping checker would encourage developers to experiment with communication structures, providing a litmus test for checking the correctness of an optimisation. While polynomial checking algorithms do exist [36], they do not support optimisation even for the simpler binary case. A potential solution lies in abstract interpretation, previously applied to *binary* subtyping, where sets of *words* model asynchronicity [3]. This addresses undecidability with systematic (lattice-theoretic) approximation. To apply abstract interpretation to multiparty interaction, we reformulate MAS as subtyping on sets of *session trees*, and make the following contributions:

- We introduce an *equivalent* but more tractable formulation of MAS [27], based on a new *lite refinement* preorder. MAS [27] is defined using nested quantification over infinite structures. One of the challenges of checking MAS is that it requires a check for *inclusion* of the actions being postponed over infinite structures. Lite refinement provides a way to automation by reducing the inclusion check to a simple, local check.
- Abstraction has been applied to binary subtyping by adapting subtyping to sets of words and then representing these sets with regular expressions [3]. MAS is formulated in terms of session trees, and likewise we generalise subtyping to sets of session trees. To derive continuations after performing an action, we adapt the Brzozowski derivative [10] from

regular expressions to session trees, and further extend it to sets of session trees. This provides a semantic foundation for subtyping, similar to a collecting semantics [17], ready for abstract interpretation and the deployment of an abstract domain.
- We provide a new abstract domain for representing sets of session trees based on type graphs [43] and provide new domain operations for send and receive. Since the abstract domain is not finite, we show how widening can be applied to derive a *terminating MAS algorithm* founded on *lite refinement*.

Overall, we show how the theoretical formulation of MAS [27] can be distilled into an algorithm for checking subtyping. Moreover, our algorithm yields a *certificate* (enabling subtyping to be double-checked by a third-party without consideration of low-level algorithmic details). We complement our theoretical contribution by evaluating our algorithm on some challenging problems and show it can prove subtyping more often than existing methods, whilst being an order of magnitude faster than the state-of-the-art binary algorithm [3].

2 Preliminaries

2.1 Syntax

Let $\mathbb{P}$ denote a finite set of roles and Σ denote a finite alphabet of communication labels. Let $\mathbb{A}^! = \{p!a \mid p \in \mathbb{P}, a \in \Sigma\}$, $\mathbb{A}^? = \{p?a \mid p \in \mathbb{P}, a \in \Sigma\}$ and $\mathbb{A} = \mathbb{A}^! \cup \mathbb{A}^?$. The syntactic category of session types, $\mathbb{S}$, is inductively defined by:

$$\mathbb{S} ::= \oplus_{i\in I}\, p!a_i.\mathbb{S}_i \mid \&_{i\in I}\, p?a_i.\mathbb{S}_i \mid \mathsf{end} \mid \mu \mathtt{t}.\mathbb{S} \mid t$$

where I is drawn from the category of (finite) index sets $\mathbb{I}$. Session type $\oplus_{i\in I}\, p!a_i.\mathbb{S}_i$ is for selection/send: the role selects one label in $\{a_i\}_{i\in I}$, sends it to role p, and continues as the corresponding $\mathbb{S}_i$. Session type $\&_{i\in I}\, p?a_i.\mathbb{S}_i$ is a branching/receive action, where the role receives a label from p. end is for termination, $\mu\mathtt{t}.\mathbb{S}$ for recursion, and $\mathtt{t}$ is a type variable used for recursive call.

Asynchronous multiparty subtyping [27] is based on a co-inductive representation of session types called *session trees*. The category $\mathbb{T}$ for session trees is defined co-inductively below. Subtyping is defined using three sub-classes of session trees called single-output (SO) trees $\mathbb{U}$, single-input (SI) trees $\mathbb{V}$, and single-input-single-output (SISO) trees $\mathbb{W}$, also defined co-inductively below:

$$\mathbb{T} ::= \oplus_{i\in I}\, p!a_i.\mathbb{T}_i \mid \&_{i\in I}\, p?a_i.\mathbb{T}_i \mid \mathsf{end} \qquad \mathbb{U} ::= p!a.\mathbb{U} \mid \&_{i\in I}\, p?a_i.\mathbb{U}_i \mid \mathsf{end}$$
$$\mathbb{V} ::= \oplus_{i\in I}\, p!a_i.\mathbb{V}_i \mid p?a.\mathbb{V} \mid \mathsf{end} \qquad \mathbb{W} ::= p!a.\mathbb{W} \mid p?a.\mathbb{W} \mid \mathsf{end}$$

We let S, T, U, V and W denote typical members $\mathbb{S}$, $\mathbb{T}$, $\mathbb{U}$, $\mathbb{V}$ and $\mathbb{W}$, respectively.

If a sequence W contains the action $p \diamond a$ where $\diamond \in \{!, ?\}$, we write $p \diamond a \in W$. Given the sets $\mathbb{Q}^! = \{p! \mid p \in \mathbb{P}\}$, $\mathbb{Q}^? = \{p? \mid p \in \mathbb{P}\}$ and $\mathbb{Q} = \mathbb{Q}^! \cup \mathbb{Q}^?$ we define the mapping $act : \mathbb{A} \to \mathbb{Q}$ where $act(p \diamond a) = p\diamond$. Given a set of actions $A \subseteq \mathbb{A}$ and a sequence of actions $W \in \mathbb{W}$, we define $act(A) = \{act(p \diamond a) \mid p \diamond a \in A\}$ and $act(W) = \{act(p \diamond a) \mid p \diamond a \in W\}$.

2.2 MAS: a declarative definition

MAS [27] is formulated in terms of two classes of sequence of actions, $\mathcal{A}^p$ and $\mathcal{B}^p$, which are parameterised by a participant p. Classes $\mathcal{A}^p$ and $\mathcal{B}^p$ embed the principles of *swapability* encoded into MAS. These *finite* sequences are inductively defined as follows:

▶ **Definition 1** (Finite sequences). *$\mathcal{A}^p ::= A^p \mid A^p.\mathcal{A}^p$ and $\mathcal{B}^p ::= B^p \mid B^p.\mathcal{B}^p$ where $A^p = \{q?a \in \mathbb{A}^? \mid p \neq q\}$ and $B^p = \{q!a \in \mathbb{A}^! \mid p \neq q\} \cup \mathbb{A}^?$.*

$\mathcal{A}^p$ defines the sequences of actions that can be anticipated before a *receive action* from p (namely any receive action that is not from p), whereas $\mathcal{B}^p$ defines the sequences of actions that can be anticipated before a *send action* to p (namely any receive action – outputs can always be done before inputs – and any send action that is not to p). Note in particular that $p?a \notin \mathcal{A}^p$ and $p!a \notin \mathcal{B}^p$. Using $\mathcal{A}^p$ and $\mathcal{B}^p$, we can now define refinement for SISO trees:

▶ **Definition 2** (SISO refinement). *The refinement relation $\precsim \subseteq \mathbb{W} \times \mathbb{W}$ for SISO-trees is co-inductively defined as below, where $A \in \mathcal{A}^p$ and $B \in \mathcal{B}^p$:*

$$\frac{}{\mathsf{end} \precsim \mathsf{end}}\ \mathsf{RefEnd} \qquad \frac{W \precsim W'}{p?a.W \precsim p?a.W'}\ \mathsf{RefIn} \qquad \frac{W \precsim W'}{p!a.W \precsim p!a.W'}\ \mathsf{RefOut}$$

$$\frac{W \precsim A.W'\ act(W) = act(A.W')}{p?a.W \precsim A.p?a.W'}\ \mathsf{RefA} \qquad \frac{W \precsim B.W'\ act(W) = act(B.W')}{p!a.W \precsim B.p!a.W'}\ \mathsf{RefB}$$

▶ **Example 3.** Since $\precsim$ is the largest relation which is closed backwards under these rules it follows $(p!a)^\omega \precsim (p!a)^\omega$ holds as well as $p!a.\mathsf{end} \precsim p!a.\mathsf{end}$. To see $(p!a)^\omega \precsim (p!a)^\omega$ observe $(p!a)^\omega = p!a.(p!a)^\omega$ therefore if $p!a.(p!a)^\omega \precsim p!a.(p!a)^\omega$ then $(p!a)^\omega \precsim (p!a)^\omega$ holds too. Thus RefOut is applicable. Rule RefB is not violated since $p!a \notin \mathcal{B}^p$ hence no rule is contradicted.

▶ **Example 4.** Let $(q?b)^\omega$ denote the infinite sequence of actions $q?b.q?b.\cdots$ and consider the following SISO trees: $W = p?a.(q?b)^\omega$ and $W' = (q?b)^\omega$. Observe that $W \precsim W'$ *cannot* be derived by RefIn since W' does not start with $p?a$ or RefA since W' does not contain $p?a$.

SISO-tree refinement provides an elegant vehicle for expressing swapability over the sequences of actions. However, a session type is not a sequence of actions, but rather a tree with selection and branching points. Definition 5 decomposes session trees in sets of SI- and SO- trees so as to express the co- and contra-variance of selection and branching.

▶ **Definition 5** (SO- and SI-tree decomposition for session trees). *The tree decomposition maps $[\![.]\!]_{\mathsf{SO}} : \mathbb{T} \to \wp(\mathbb{U})$ and $[\![.]\!]_{\mathsf{SI}} : \mathbb{T} \to \wp(\mathbb{V})$ are defined:*

$$\begin{array}{ll}
[\![\mathsf{end}]\!]_{\mathsf{SO}} = \{\mathsf{end}\} & [\![\mathsf{end}]\!]_{\mathsf{SI}} = \{\mathsf{end}\} \\
[\![\bigoplus_{i\in I} p!a_i.T_i]\!]_{\mathsf{SO}} = \{p!a_i.U_i \mid i \in I, U_i \in [\![T_i]\!]_{\mathsf{SO}}\} & [\![\bigoplus_{i\in I} p!a_i.T_i]\!]_{\mathsf{SI}} = \{\bigoplus_{i\in I} p!a_i.V_i \mid i \in I, V_i \in [\![T_i]\!]_{\mathsf{SI}}\} \\
[\![\&_{i\in I} p?a_i.T_i]\!]_{\mathsf{SO}} = \{\&_{i\in I} p?a_i.U_i \mid i \in I, U_i \in [\![T_i]\!]_{\mathsf{SO}}\} & [\![\&_{i\in I} p?a_i.T_i]\!]_{\mathsf{SI}} = \{p?a_i.V_i \mid i \in I, V_i \in [\![T_i]\!]_{\mathsf{SI}}\}
\end{array}$$

The intuition is that SO-tree decomposition removes the selection points in a session tree and expresses them using *sets* of single-output sequences, and dually for SI-trees. The maps $[\![\cdot]\!]_{\mathsf{SO}}$ and $[\![\cdot]\!]_{\mathsf{SI}}$ lift to a set of session trees $\mathcal{T} \in \wp(\mathbb{T})$ by, respectively, $[\![\mathcal{T}]\!]_{\mathsf{SO}} = \cup_{T\in\mathcal{T}} [\![T]\!]_{\mathsf{SO}}$ and $[\![\mathcal{T}]\!]_{\mathsf{SI}} = \cup_{T\in\mathcal{T}} [\![T]\!]_{\mathsf{SI}}$.

▶ **Example 6.** $[\![U]\!]_{\mathsf{SI}} = \{(p?a)^\omega\} \cup \{(p?a)^k.p?b.q?b.\mathsf{end} \mid k \geq 0\}$ where $U = p?a.U \,\&\, p?b.q?b.\mathsf{end}$.

Multiparty Asynchronous Subtyping (MAS) is formally defined as a binary relation on session trees in a forall-exists construction. Covariance and contravariance are encoded as a forall requirement on the SO-trees (resp. SI-trees) of the subtype (resp. supertype). The problem is then reduced to checking SISO-tree refinement:

▶ **Definition 7** (MAS). *The binary relation $\leq$ on $\mathbb{T} \times \mathbb{T}$ is defined by $T \leq T'$ holds iff $\forall U \in [\![T]\!]_{\mathsf{SO}} : \forall V' \in [\![T']\!]_{\mathsf{SI}} : \exists W \in [\![U]\!]_{\mathsf{SI}} : \exists W' \in [\![V']\!]_{\mathsf{SO}} : W \precsim W'$.*

▶ **Example 8.** To illustrate MAS, consider showing $T \leq T'$ where $T = p?c.q!d.\mathsf{end} \,\&\, p?e.q!d.\mathsf{end}$ and $T' = q!d.(p?c.\mathsf{end} \,\&\, p?e.\mathsf{end})$. Since T is an SO-tree, $[\![T]\!]_{\mathsf{SO}} = \{T\}$. T' is not an SI-tree hence $[\![T']\!]_{\mathsf{SI}} = \{q!d.p?c.\mathsf{end}, q!d.p?e.\mathsf{end}\}$. Universal quantification over $[\![T']\!]_{\mathsf{SI}}$ requires that T is checked against *both* $q!d.p?c.\mathsf{end}$ and $q!d.p?e.\mathsf{end}$. To do so, note $p?c.q!d.\mathsf{end} \in [\![T]\!]_{\mathsf{SI}}$ and $p?c.q!d.\mathsf{end} \precsim q!d.p?c.\mathsf{end}$ likewise $p?e.q!d.\mathsf{end} \in [\![T]\!]_{\mathsf{SI}}$ and $p?e.q!d.\mathsf{end} \precsim q!d.p?e.\mathsf{end}$ since

$$\dfrac{\dfrac{\dfrac{}{\mathsf{end} \precsim \mathsf{end}}\,\mathsf{RefEnd}}{q!d.\mathsf{end} \precsim q!d.\mathsf{end}}\,\mathsf{RefOut}}{p?c.q!d.\mathsf{end} \precsim q!d.p?c.\mathsf{end}}\,\mathsf{RefA} \qquad \dfrac{\dfrac{\dfrac{}{\mathsf{end} \precsim \mathsf{end}}\,\mathsf{RefEnd}}{q!d.\mathsf{end} \precsim q!d.\mathsf{end}}\,\mathsf{RefOut}}{p?e.q!d.\mathsf{end} \precsim q!d.p?e.\mathsf{end}}\,\mathsf{RefA}$$

3 Lite SISO-tree refinement

To develop a subtyping algorithm we introduce a novel form of SISO-refinement on $\mathbb{W} \times \mathbb{W}$, called lite SISO-tree refinement, that is **equivalent to classical SISO-refinement** but is more attractive computationally. Lite refinement builds on the observation that the equality checks $act(W) = act(A.W')$ and $act(W) = act(B.W')$ in the premises RefA and RefB of classical SISO-refinement can be relaxed to inclusion checks $act(W) \subseteq act(A.W')$ and $act(W) \subseteq act(B.W')$ without enlarging the refinement relation. Furthermore, the inclusions can be further relaxed to lightweight checks $act(hd(A)) \in act(W)$ and $act(hd(B)) \in act(W)$ involving only the single actions $hd(A)$ and $hd(B)$ where hd denotes head:

▶ **Definition 9.** *The lite refinement relation $\precsim_l \subseteq \mathbb{W} \times \mathbb{W}$ on SISO-trees is co-inductively defined as:*

$$\dfrac{W \precsim_l A.W' \; act(hd(A)) \in act(W)}{p?a.W \precsim_l A.p?a.W'}\,\mathsf{RefA} \qquad \dfrac{W \precsim_l B.W' \; act(hd(B)) \in act(W)}{p!a.W \precsim_l B.p!a.W'}\,\mathsf{RefB}$$

where $A \in \mathcal{A}^p$ and $B \in \mathcal{B}^p$ with RefEnd, RefIn *and* RefOut *as in Definition 2.*

▶ **Corollary 10.** *$W \precsim W'$ iff $W \precsim_l W'$.*

In the sequel, we develop a subtyping algorithm, based on $\precsim_l$, the correctness of which follows from the equivalence asserted by Corollary 10.

4 Subtyping Sets of Session Trees

Reasoning on *sets* of session trees paves the way for abstract interpretation [17] to be applied, where an algorithm is obtained by representing sets of session trees using an *abstract domain* constructed from type graphs [32]. Let $[\![\mathcal{T}']\!]_{\mathsf{SI}} = \cup\{[\![T']\!]_{\mathsf{SI}} \mid T' \in \mathcal{T}'\}$ to faciliate the following generalisaton of classical subtyping [27] to sets of session trees:

▶ **Definition 11.** *The binary relation $\leq \subseteq \mathbb{T} \times \wp(\mathbb{T})$ is defined $T \leq \mathcal{T}'$ holds iff $\mathcal{T}' \neq \emptyset$ and $\forall\, U \in [\![T]\!]_{\mathsf{SO}}$: $\forall\, V' \in [\![\mathcal{T}']\!]_{\mathsf{SI}}$: $\exists\, W \in [\![U]\!]_{\mathsf{SI}}$: $\exists W' \in [\![V']\!]_{\mathsf{SO}}$: $W \precsim W'$.*

The section builds towards a subtyping relation on sets of subtrees, given in Figure 1, that provides a basis for abstract interpretation. Subtyping is defined in terms two notions of derivative: one for receive actions (Definition 16) and one for send actions (Definition 17). These operations scan a session tree from its root for a specific action, and return the session trees obtained by removing that action and any sub-trees that do not include it. These operations are inspired by the Brzozowski derivative [10] of a regular expression

e wrt to a symbol u: if e represents a set of words W, the derivative $\partial_u(e)$ represents $\{w \in \Sigma^* \mid u \cdot w \in W\}$. Our derivatives, presented in Sections 4.1 and 4.2, generalise the Brzozowski derivative to operate on (sets of) session trees rather than (sets of) words. Section 4.3 formulates subtyping using the derivatives. The subtyping relation is defined co-inductively, each deduction reading the action at the head of the subtype to apply either the send or receive derivative to the supertype. Our approach is justified by lite refinement.

4.1 Receive derivative of a session tree

Derivatives differ depending on whether they look for receive or send actions. A receive derivative is calculated using a depth bound derived using the $\mathsf{always}_{p?}$ predicate:

▶ **Definition 12.** *The predicate* $\mathsf{always}_{p?} \subseteq \mathbb{T} \times \mathbb{N}_0$ *is inductively defined as follows:*

$$\frac{}{\mathsf{always}_{p?}(\&_{i\in I} p?a_i.T'_i, 0)} \qquad \frac{p \neq q \quad \forall i \in I : \mathsf{always}_{p?}(T'_i, k_i) \quad k = 1 + \max_{i\in I} k_i}{\mathsf{always}_{p?}(\&_{i\in I} q?a_i.T'_i, k)} \qquad \frac{\forall i \in I : \mathsf{always}_{p?}(T'_i, k_i) \quad k = 1 + \max_{i\in I} k_i}{\mathsf{always}_{p?}(\bigoplus_{i\in I} q!a_i.T'_i, k)}$$

Furthermore $\mathsf{always}_{p?}(T)$ *holds iff there exists* $k \in \mathbb{N}_0$ *such that* $\mathsf{always}_{p?}(T, k)$ *holds.*

▶ **Example 13.** Let $T'_1 = q?a.p?a.\mathsf{end} + q?b.p?b.T'_1$ and $T'_2 = q?a.p?a.\mathsf{end} + q?b.T'_2$. Since $\mathsf{always}_{p?}(p?a.\mathsf{end}, 0)$ and $\mathsf{always}_{p?}(p?b.T'_1, 0)$ then $\mathsf{always}_{p?}(T'_1, 1)$. However, $\mathsf{always}_{p?}(T'_2, k)$ does not hold $(\forall\, k \in \mathbb{N}_0)$ since $\mathsf{always}_{p?}$ is defined inductively.

Lemma 14 asserts that the absence of a bound k implies the absence of subtyping. Lemma 15 asserts that there exists at most one k for which $\mathsf{always}_{p?}(T', k)$ holds. The receive derivative, which follows, is then defined in terms of the bound.

▶ **Lemma 14.** *If* $T, T' \in \mathbb{T}$ *and* $p?a.T \leq T'$ *then* $\mathsf{always}_{p?}(T')$.

▶ **Lemma 15.** *If* $T' \in \mathbb{T}$, $\mathsf{always}_{p?}(T', k)$ *and* $\mathsf{always}_{p?}(T', k')$ *then* $k = k'$.

▶ **Definition 16** (Receive derivative)**.** *The receive derivative map* $\delta_{p?a} : \mathbb{T} \to \wp(\mathbb{T})$ *is defined as* $\delta_{p?a}(T) =$ *if* $\mathsf{always}_{p?}(T, k)$ *then* $\{D \mid \delta^k_{p?a}(T, D)\}$ *else* $\emptyset$*, where the derivative relation* $\delta^k_{p?a} : \mathbb{T} \times \mathbb{T}$ *for some* $k \geq 0$ *is inductively defined:*

$$\frac{j \in I \quad a = a_j}{\delta^0_{p?a}(\&_{i\in I} p?a_i.T'_i, T'_j)} \qquad \frac{p \neq q \quad \forall i \in I : \delta^{k_i}_{p?a}(T'_i, D_i) \quad k = 1 + \max\{k_i\}_{i\in I} \quad j \in I}{\delta^k_{p?a}(\&_{i\in I} q?a_i.T'_i, q?a_j.D_j)}$$

Following MAS, a receive action from p can only be found immediately (first rule) or after other receive actions from other roles (second rule). The second rule can be applied non-deterministically for any $j \in I$ and derives at least $|I|$ distinct derivatives (which $\delta_{p?a}(T)$ returns as a set). We write $q?a_j.D_j$ for the concatenation of $q?a$, the action preceding $p?a$, with a subtree D_j that was the continuation of $p?a$ in that path. The receive derivative is well-defined because there exists at most one k such that $\mathsf{always}_{p?}(T', k)$ holds.

4.2 Send derivative of a session tree

Unfortunately, Lemma 14 does not help us define a send derivative on supertypes as we no longer rely on a known bound k. Hence, the definition of $\Delta^k_{p!a}$ is parametric in k.

▶ **Definition 17** (Send derivative). *The k-th order send derivative map $\Delta^k_{p!a} : \mathbb{T} \to \wp(\mathbb{T})$ is defined as*

$$\Delta^k_{p!a}(T') = \begin{cases} \mathcal{D}^k & \text{if } \mathcal{D}^k \neq \emptyset \ \vee \ k = 0 \qquad \text{where } \mathcal{D}^k = \{D \mid \Delta^k_{p!a}(T', D)\} \\ \Delta^{k-1}_{p!a}(T') & \text{otherwise} \end{cases}$$

and the derivative relation $\Delta^k_{p!a} \subseteq \mathbb{T} \times \mathbb{T}$ is, itself, inductively defined by:

$$\frac{j \in I \quad a = a_j}{\Delta^0_{p!a}(\bigoplus_{i\in I} p!a_i.T'_i, T'_j)} \qquad \frac{\forall i \in I : \Delta^{k_i}_{p!a}(T'_i, D_i) \wedge k_i \text{ maximal} \quad k = 1 + \max\{k_i\}_{i\in I} \quad j \in I}{\Delta^k_{p!a}(\&_{i\in I} q?a_i.T'_i, q?a_j.D_j)}$$

$$\frac{p \neq q \quad J = \{i \in I \mid \Delta^{k_i}_{p!a}(T'_i, D_i) \ \wedge \ k_i \text{ maximal } \} \neq \emptyset \quad k = 1 + \max\{k_i\}_{i\in I}}{\Delta^k_{p!a}(\bigoplus_{i\in I} q!a_i.T'_i, \bigoplus_{j\in J} q!a_j.D_j)}$$

The k-th derivative is defined to be the set of derivatives $\mathcal{D}^\ell$ for which $\mathcal{D}^\ell \neq \emptyset$, the degree ℓ is maximal and $\ell \leq k$. If $\mathcal{D}^\ell = \emptyset$ for all $0 \leq \ell \leq k$ then the k-th derivative is itself null. Following MAS, in a supertype, a send action can be found immediately (first rule), after receive actions (e.g., the subtype may have anticipated the output), or after other send actions from different roles (third rule). Condition k_i maximal means: $\forall_{i\in I} : \Delta^{k^*_i}_{p!a}(T'_i, D^*_i) \Rightarrow k^*_i \leq k_i$. The third rule derives at least one derivative equipped with $|J|$ branches $q!a_j.D_j$, where the D_j are again selected to maximise their degree k_i.

▶ **Example 18** (Derivatives). Let $T = r?a.q?a.p!a.T \& r?b.r?b.q?a.\mathsf{end}$. The receive derivative $\delta_{q?a}(T)$ exists because both branches of T possess a receive action $q?a$. Indeed $\mathsf{always}_{q?}(T, 2)$ holds, hence $\delta_{q?a}(T) = \{r?a.p!a.T,\ r?b.r?b.\mathsf{end}\}$. Now consider $\Delta^k_{p!a}(T)$. For $k < 2$, the send derivatives $\Delta^0_{p!a}(T) = \Delta^1_{p!a}(T) = \emptyset$. Yet for $k \geq 2$, $\Delta^k_{p!a}(T) = \{r?a.q?a.T\}$. In this case, $p!a$ does not need to occur in all branches of T for the derivative to exist.

4.3 Subtyping with receive and send derivatives

To apply one derivative after another in a chain of actions, receive and send derivatives are lifted to operators on sets of supertypes as follows:

▶ **Definition 19.** *The operators $\delta_{p?a} : \wp(\mathbb{T}) \to \wp(\mathbb{T})$ and $\Delta_{p!a} : \wp(\mathbb{T}) \to \wp(\mathbb{T})$ are defined:*

$$\begin{array}{l} \delta_{p?a}(\mathcal{T}') = \cup\{\delta_{p?a}(T') \mid T' \in \mathcal{T}'\} \\ \Delta^k_{p!a}(\mathcal{T}') = \uplus\{\Delta^k_{p!a}(T') \mid T' \in \mathcal{T}'\} \end{array} \quad \text{where} \quad \uplus\,\mathcal{T}' = \begin{cases} \emptyset & \emptyset \in \mathcal{T}' \\ \cup\mathcal{T}' & \text{otherwise} \end{cases}$$

The use of $\uplus$ in $\Delta^k_{p!a}(\mathcal{T}')$ reflects covariance over sends: the $p!a$ action of the subtype must be supported by all session-trees $T' \in \mathcal{T}'$. Conversely, the use of $\cup$ in $\delta_{p?a}(\mathcal{T}')$ models contravariance over receives: the $p?a$ of the subtype does not need to be found in all $T' \in \mathcal{T}'$.

▶ **Definition 20.** *Given a partial map $\Gamma \in \mathbb{T} \rightharpoonup \wp(\mathbb{Q})$, the update and accumulate operations are respectively defined:*

- $\Gamma[T \mapsto Q] = \{T \mapsto Q\} \cup \{T' \mapsto Q' \in \Gamma \mid T \neq T'\}$
- $\Gamma + p\diamond = \{T \mapsto Q \cup \{p\diamond\} \mid T \mapsto Q \in \Gamma\}$ *where* $\diamond \in \{!, ?\}$

▶ **Definition 21** (Subtyping on sets of session trees). *The ternary subtyping relation $\leq \subseteq (\mathbb{T} \rightharpoonup \wp(\mathbb{Q})) \times \mathbb{T} \times \wp(\mathbb{T})$ is co-inductively defined by the rules of Figure 1.*

The ternary subtyping relation is defined using an environment Γ, a partial map $\mathbb{T} \rightharpoonup \wp(\mathbb{Q})$, that records the actions performed by T before T is encountered again as a sub-tree. This device assumes that session trees are regular [40, Section 21.7.2] (aka rational [16]). T is

$$\frac{}{\Gamma \vdash \mathsf{end} \leq \{\mathsf{end}\}}\ \mathsf{RefEnd}$$

$$\frac{\begin{array}{c}\forall T' \in \mathcal{T}' : \exists i \in I : \delta_{p?a_i}(T') \neq \emptyset \qquad J = \{i \in I \mid \delta_{p?a_i}(\mathcal{T}') \neq \emptyset\} \neq \emptyset \\ \forall i \in J : \exists \mathcal{D}_i \supseteq \delta_{p?a_i}(\mathcal{T}) : \Gamma' \vdash T_i \leq \mathcal{D}_i \\ \hdashline T \mapsto Q \in \Gamma \Rightarrow Q \supseteq act(hd(\mathcal{T}')) \quad \Gamma' = \Gamma[T \mapsto \emptyset] + p? \quad \text{where} \quad T = \&_{i\in I} p?a_i.T_i\end{array}}{\Gamma \vdash \&_{i\in I} p?a_i.T_i \leq \mathcal{T}'}\ \mathsf{RefInA}$$

$$\frac{\begin{array}{c}\exists k \in \mathbb{N} : \forall i \in I : \Delta^k_{p!a_i}(\mathcal{T}') \neq \emptyset \qquad \forall i \in I : \exists \mathcal{D}_i \supseteq \Delta^k_{p!a_i}(\mathcal{T}) : \Gamma' \vdash T_i \leq \mathcal{D}_i \\ \hdashline T \mapsto Q \in \Gamma \Rightarrow Q \supseteq act(hd(\mathcal{T}')) \quad \Gamma' = \Gamma[T \mapsto \emptyset] + p! \quad \text{where} \quad T = \oplus_{i\in I} p!a_i.T_i\end{array}}{\Gamma \vdash \oplus_{i\in I} p!a_i.T_i \leq \mathcal{T}'}\ \mathsf{RefOutB}$$

Figure 1 Rules for the ternary subtyping relation $\leq \subseteq (\mathbb{T} \rightharpoonup \wp(\mathbb{Q})) \times \mathbb{T} \times \wp(\mathbb{T})$.

regular iff its set of distinct subtrees, $\mathsf{sub}(T)$, is finite. This is not limiting since session types correspond to regular session trees. The premises of RefInA and RefOutB are partitioned by the dotted lines into: (bottom) requirements based on the syntax of the subtype T that decompose the problem onto sub-problems of the form $\Gamma_i \vdash T_i \leq \mathcal{D}_i$ and ensure a *progress condition*, and (top) subtyping requirements based on derivatives. In both rules, the premise $T \mapsto Q \in \Gamma \Rightarrow Q \supseteq act(hd(\mathcal{T}'))$ stipulates that the actions at the head of $\mathcal{T}'$ must be performed within one cycle of T, that is, before T is revisited while traversing the subtype. This progress condition [15] ensures that a head action of $\mathcal{T}'$ is not postponed indefinitely.[2]

Rule RefInA lifts RefIn and RefA of lite refinement (Definition 9) to sets of session trees. The first premise of RefInA (top) is a coverage condition which ensures that for every $T' \in \mathcal{T}'$ a receive derivative $\delta_{p?a_i}(T')$ is defined for least one $i \in I$. This guarantees that the selection of receive actions offered by T cover those occurring in $\mathcal{T}'$.

Rule RefOutB lifts RefOut and RefB (Definition 9) to session trees by applying the send derivative. Note RefInA permits a relaxation $\mathcal{D}_i$ to be used for $\delta_{p?a_i}(\mathcal{T}')$; likewise a superset $\mathcal{D}_i$ is used for $\Delta^{k_i}_{p!a_i}(\mathcal{T}')$ in RefOutB. Observe k is not fixed upfront: it can vary with $\mathcal{T}'$.

▶ **Theorem 22.** *If $T \in \mathbb{T}$ is regular, $\mathcal{T}' \in \wp(\mathbb{T})$ and $\emptyset \vdash T \leq \mathcal{T}'$ then $T \leq \mathcal{T}'$.*

▶ **Example 23** (Violated progress condition)**.** Consider $U = p?a.(q?a.U + q?b.r?a.\mathsf{end})$ and the subtyping problem $U \leq r?a.W'$ where $W' = p?a.q?a.W'$ and $W' \in \mathbb{W}$. Observe that if $W \precsim_l r?a.W'$ then $r? \in act(W)$ hence W is finite which presents a contradiction therefore $U \not\leq r?a.W'$. Likewise $\emptyset \vdash U \not\leq r?a.W'$ as is demonsrated below:

$$\dfrac{\dfrac{\{U \mapsto \{p?, q?\}, q?a.U \mapsto \{q?\}\} \vdash U \leq \{r?a.W'\}}{\{U \mapsto \{q?\}\} \vdash q?a.U + q?b.r?a.\mathsf{end} \leq \{r?a.q?a.W'\}}\ \mathsf{RefInA}}{\emptyset \vdash U \leq \{r?a.W'\}}\ \mathsf{RefInA}$$

In the first application of RefInA, $\delta_{p?a}(r?a.W') = \delta_{p?a}(r?a.p?a.q?a.W') = \{r?a.q?a.W'\}$ and $\delta_{p?a'}(r?a.W') = \emptyset$ for all $a' \neq a$. In the second, $\delta_{q?a}(r?a.q?a.W') = \{r?a.W'\}$ but $\delta_{q?b}(r?a.q?a.W') = \emptyset$ thus J is a singleton. The development cannot be progressed since $\{p?, q?\} \not\supseteq \{r?\} = act(hd(\{r?a.W'\}))$, in effect, detecting that $r?a$ is an orphan.

[2] Binary subtyping [3, 5, 6, 13] makes use of a check that tests for cycles of send actions in the candidate subtype. We reframed these checks here as a more intuitive and simple progress condition, which amounts to no more than a set inclusion test that discharges a requirement on swapability.

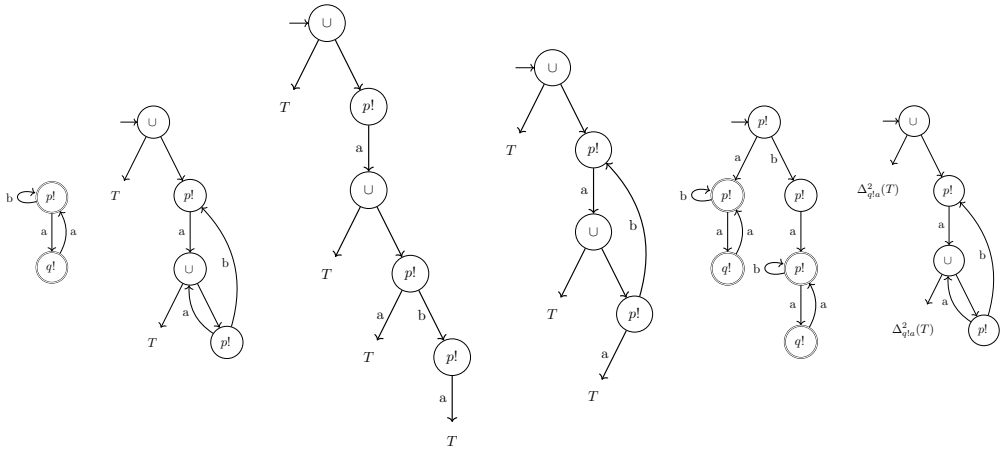


Figure 2 (i) T (ii) G (iii) W_0 (iv) W_1 (v) $\Delta^2_{q!a}(T)$ (vi) $\Pi^2_{q!a}(G)$.

▶ **Example 24** (Satisfied progress conditions)**.** To illustrate where a message is postponed, but is not orphan, consider the question $W \leq W'$ where $W = p!a.p!a.p?b.W$ and $W' = p!a.p?b.W'$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\{W \mapsto \{p!\}, p!a.p?b.W \mapsto \{p!,p?\}, p?b.W \mapsto \{p?,p!\}\} \vdash p!a.p?b.W \leq \{p?b.p?b.W'\}}\,\mathsf{RefOutB}}{\{W \mapsto \{p!,p?\}, p!a.p?b.W \mapsto \{p!,p?\}, p?b.W \mapsto \{p?\}\} \vdash W \leq \{p?b.W'\}}\,\mathsf{RefInA}}{\{W \mapsto \{p!\}, p!a.p?b.W \mapsto \{p!\}\} \vdash p?b.W \leq \{p?b.p?b.W'\}}\,\mathsf{RefOutB}}{\{W \mapsto \{p!\}\} \vdash p!a.p?b.W \leq \{p?b.W'\}}\,\mathsf{RefOutB}}{\emptyset \vdash W \leq \{W'\}}$$

To show $\emptyset \vdash W \leq \{W'\}$, it is sufficient to put $k = 0$ since $\Delta^0_{p!a}(W') = \{p?b.W'\}$. To establish the second, it is necessary to use $k = 1$ because $\Delta^0_{p!a}(p?b.W') = \emptyset$ but $\Delta^1_{p!a}(p?b.W') = \{p?b.p?b.W'\}$. To show the fourth, $k = 2$ is needed, illustrating that it is not sufficient to fix k throughout. Notice too that W is first revisited when attempting to establish the the fourth judgement. Its environment includes $W \mapsto \{p!, p?\} \in \Gamma$ and $\{p?\} = act(hd(\{p?b.W'\}))$ hence the environment (orphan) premise is satisfied, and the development continues.

5 Subtyping sets of session trees using type graphs

Type graphs [32, Section 4.1.2] were proposed for representing (possibly infinite) sets of finite trees. We adapt them to represent (possibly infinite) sets of session trees by inductively defining the category $\mathbb{G}$ of type graphs to be: $\mathbb{G} ::= \emptyset \mid \mathbb{T} \mid \cup_{i\in I}\mathbb{G}_i \mid \&_{i\in I}p?a_i.\mathbb{G}_i \mid \oplus_{i\in I}p!a_i.\mathbb{G}_i$. Thus the leaves of a type graph G are session trees drawn from $\mathbb{T}$. A type graph is interpreted by a *concretisation* map γ [17] which describes the set of session trees it represents.

▶ **Definition 25** (Concretisation map)**.** *The concretisation map $\gamma : \mathbb{G} \to \wp(\mathbb{T})$ is defined as the least solution to the following equations:*

$$\begin{array}{rl} \gamma(\emptyset) = \emptyset & \gamma(\&_{i\in I}p?a_i.G_i) = \{\&_{i\in I}p?a_i.T_i \mid \forall i \in I : T_i \in \gamma(G_i)\} \\ \gamma(T) = \{T\} & \gamma(\oplus_{i\in I}p!a_i.G_i) = \{\oplus_{i\in I}p!a_i.T_i \mid \forall i \in I : T_i \in \gamma(G_i)\} \\ \gamma(\cup_{i\in I}G_i) = \cup_{i\in I}\gamma(G_i) & \end{array}$$

▶ **Example 26.** Consider T (co-inductively) defined by $T = p!a.q!a.T \oplus p!b.T$ and the type graph $G = T \cup p!a.G'$ where G' is (inductively) defined by $G' = T \cup (p!a.G' \oplus p!b.p!a.G')$. T and G are illustrated in Figure 2 (i) and (ii) as regular trees where the terminal symbols distinguish T from G. If $\mathcal{T}_0 = \{T\}$ and $\mathcal{T}_{i+1} = \{T', p!a.T', p!a.(p!a.T' \oplus p!b.p!a.T') \mid T' \in \mathcal{T}_i\}$ then $\gamma(G) = \cup_{i=0}^{\infty}\mathcal{T}_i$. Observe that G has a finite representation even though $\gamma(G)$ is infinite.

Concretisation induces a preorder $\preceq$ on $\mathbb{G}$ by $G \preceq G'$ iff $\gamma(G) \subseteq \gamma(G')$. The preorder $\langle \mathbb{G}, \preceq \rangle$ may possess a bottom element $\emptyset$ but admits infinite ascending chains, as shown below:

▶ **Example 27.** Define $G_0 = T$ and $G_{i+1} = G_i \cup p!a.(G_i \cup p!a.G_i \oplus p!b.p!a.G_i)$. Observe $\gamma(G_i) \subseteq \gamma(G_{i+1})$ but $\gamma(G_i) \neq \gamma(G_{i+1})$ for all $i \geq 0$. Nevertheless the chain G_i has an upper bound: G of Example 26 since $\gamma(G_i) \subseteq \gamma(G)$ for all $i \geq 0$.

As we shall see, calculating the upper bound of a sequence of type graphs is key to finitely computing a co-inductive proof. Such an upper bound can classically [17] be found with a widening operator $\mathbb{G} \triangledown \mathbb{G} \to \mathbb{G}$. Widening provides a way to finitely derive an upper bound: given any increasing sequence $G_0 \preceq G_1 \cdots$ the (widened) sequence defined by $G'_0 = G_0$ and $G'_{i+1} = G'_i \triangledown G_{i+1}$ is a sequence $G'_0 \preceq G'_1 \preceq \cdots$ which is ultimately stable with a limit G'_ℓ. Widening is required [17] to be monotonic in both arguments: $G \preceq G \triangledown G'$ and $G' \preceq G \triangledown G'$. Then the limit G'_ℓ is an upper bound of the whole G_i sequence, that is, $G_i \preceq G'_\ell$ for all $i \geq 0$.

▶ **Example 28.** Type graphs can be widened [43] by introducing back-arcs to curb growth and thereby induce stability. To provide some intuition, consider widening the type graph W_0 given in Figure 2(iii). Widening is achieved by repeatedly introducing cycles into the type graph by replacing a sub-tree with a back-arc to an ancestor. To illustrate, W_0 is transformed into W_1, see Figure 2(iv), by replacing the sub-tree $p!a.T$ with a back-arc to an ancestor. Observe $W_0 \preceq W_1$. Moreover, the deepest T of W_1 has a direct ancestor which is disjunctive and has T as a child. This triggers the introduction of another back-arc which finally yields G, see Figure 2(ii). Again $W_1 \preceq G$. The widening algorithm of [43] guarantees [43, Theorem 7.1] that a widened sequence is ultimately stable, while attempting to minimise the introduction of cycles. The widening of [43] will terminate with G, though a more aggressive widening would also replace the deepest T of G with a back-arc to the root.

5.1 An abstract domain for sets of session trees

Operations on type graphs can be designed to mimic those on sets of session trees. In particular, to faithfully mimic the derivatives, it is both necessary to detect the absence of a derivative and preserve the derivation in a relaxation. The following definition formulates these requires and the proposition shows how they can be satisfied by lifting $\Delta^k_{p!a}$ and $\delta_{p?a}$:

▶ **Definition 29.** *An abstract send derivative is any operator $\Pi^k_{p!a} : \mathbb{G} \to \mathbb{G}$ such that:*

- *if $\Delta^k_{p!a}(\gamma(G)) = \emptyset$ then $\Pi^k_{p!a}(G) = \emptyset$*
- *$\Delta^k_{p!a}(\gamma(G)) \subseteq \gamma(\Pi^k_{p!a}(G))$*

for all $G \in \mathbb{G}$. An abstract receive derivative $\pi_{p?a} : \mathbb{G} \to \mathbb{G}$ is specified analogously.

▶ **Proposition 30.** *The operator $\mathbb{G} \to \mathbb{G}$ is an abstract send derivative where:*

$$\frac{}{\Gamma \vdash \Pi^k_{p!a}(\emptyset) = \emptyset} \qquad \frac{G \in \mathbb{T} \quad \Delta^k_{p!a}(G) = \emptyset}{\Gamma \vdash \Pi^k_{p!a}(G) = \emptyset} \qquad \frac{G \in \mathbb{T} \quad \Delta^k_{p!a}(G) = \{T_i\}_{i\in I}}{\Gamma \vdash \Pi^k_{p!a}(G) = \cup_{i\in I} T_i} \qquad \frac{(\cup_{i\in I} G_i) \mapsto G \in \Gamma}{\Gamma \vdash \Pi^k_{p!a}(\cup_{i\in I} G_i) = G}$$

$$\frac{\cup_{i\in I} G_i \notin \mathsf{dom}(\Gamma) \quad \Gamma' = \Gamma \cup \{\cup_{i\in I} G_i \mapsto \uplus_{i\in I} G'_i\} \quad \forall i \in I : \Gamma' \vdash \Pi^k_{p!a}(G_i) = G'_i}{\Gamma \vdash \Pi^k_{p!a}(\cup_{i\in I} G_i) = \uplus_{i\in I} G'_i}$$

with additional rules for $\Pi^k_{p!a}(\&_{i\in I} p?a_i.G_i)$ and $\Pi^k_{p!a}(\oplus_{i\in I} p!a_i.G_i)$ that follow Definition 17.

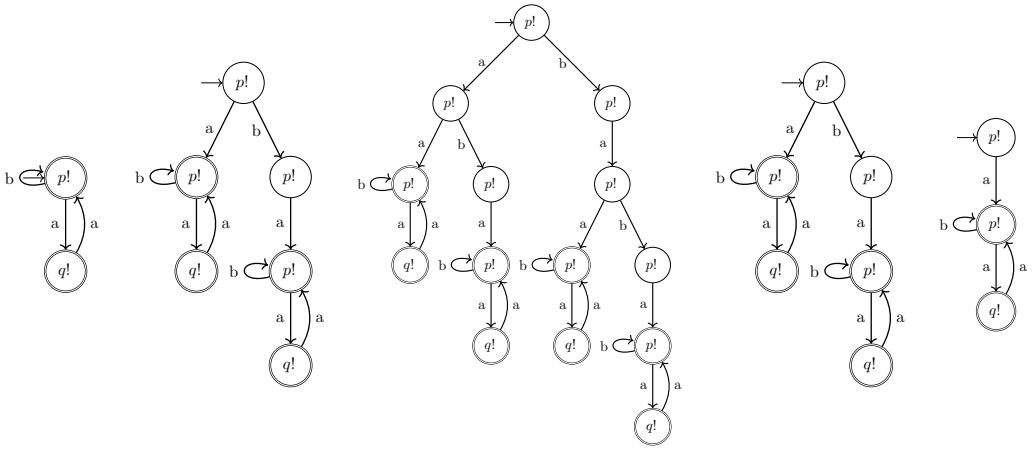


Figure 3 Intermediate type graphs: (i) $G'_{0,0}$ (ii) $G_{0,1}$ (iii) $G_{0,2}$ (iv) $G_{0,3}$ and (v) $G_{1,0}$.

An abstract receive derivative $\pi_{p?a} : \mathbb{G} \to \mathbb{G}$ can also be constructed using an environment Γ.

▶ **Example 31.** Consider calculating $\Pi^2_{q!a}(G)$ where G is as defined in Example 26. Figure 2 (i) and (ii) illustrate T and G. Following Proposition 30, $\Pi^2_{q!a}(G)$ is constructed by replacing each leaf T of G with the derivative $\Delta^2_{q!a}(T)$, given in (v), to produce the type graph of (vi).

An algorithm for checking subtyping follows from Figure 1 by substituting $\Delta^k_{p!a}$ with $\Pi^k_{p!a}$ and $\delta_{p?a}$ with $\pi_{p?a}$; thus operations on sets of session trees are simulated with operations on type graphs. We provide insight by first showing how widening can be applied to type graphs to create a co-inductive proof of subtyping, for a subtype that is linearly recursive.

▶ **Example 32.** To illustrate our strategy, consider $T \leq Q$ where $T = q!a.q!a.p!a.p!b.T$ and $Q = p!a.q!a.Q \oplus p!b.Q$. For brevity, define $T_0 = q!a.T_1$, $T_1 = q!a.T_2$, $T_2 = p!a.T_3$ and $T_3 = p!b.T_0$ thus $T = T_0$. Suppose $G'_{0,0} = Q$ and $G'_{i+1,0} = G'_{i,0} \cup G_{i+1,0}$ for all $i \geq 0$ where

$$\frac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma'_0 \vdash T_0 \leq G_{1,0}}{\Gamma'_3 \vdash T_3 \leq G_{0,3}}}{\Gamma'_2 \vdash T_2 \leq G_{0,2}}}{\Gamma'_1 \vdash T_1 \leq G_{0,1}}}{\Gamma'_0 \vdash T_0 \leq G'_{0,0}} \quad \frac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma'_0 \vdash T_0 \leq G_{i+1,0}}{\Gamma'_3 \vdash T_3 \leq G_{i,3}}}{\Gamma'_2 \vdash T_2 \leq G_{i,2}}}{\Gamma'_1 \vdash T_1 \leq G_{i,1}}}{\Gamma'_0 \vdash T_0 \leq G'_{i,0}} \quad \frac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma'_0 \vdash T_0 \leq G_{\ell+1,0}}{\Gamma'_3 \vdash T_3 \leq G_{\ell,3}}}{\Gamma'_2 \vdash T_2 \leq G_{\ell,2}}}{\Gamma'_1 \vdash T_1 \leq G_{\ell,1}}}{\Gamma'_0 \vdash T_0 \leq G'_{\ell,0}}$$

Observe $G'_{0,0} \preceq G'_{1,0} \preceq \cdots$ constitutes an increasing sequence, albeit one that is possibility infinite. Repeating the construction with $G'_{i+1,0} = G'_{i,0} \triangledown G_{i+1,0}$, however, yields a sequence which is ultimately stable, that contains its limit $G'_{\ell,0}$. Because $G'_{\ell,0} \triangledown G_{\ell+1,0} = G'_{\ell+1,0} = G'_{\ell,0}$ it follows $G_{\ell+1,0} \preceq G'_{\ell,0}$. An infinite co-inductive derivation can then be assembled by repeating the derivation on the right (since RefInA and RefOutB support relaxation). Figure 3 illustrates $G'_{0,0}$, $G_{0,1}$, $G_{0,2}$, $G_{0,3}$ and $G_{1,0}$ whereas Figure 4 gives the widened sequence $G'_{0,0} \preceq G'_{1,0} \preceq G'_{2,0} \preceq G'_{3,0} \cdots$ which has $G'_{3,0}$ as its limit. Altogether this shows $T \leq Q$. Note that $G'_{3,0}$ constitutes a certificate. Finally observe that existence of $G'_{3,0}$, hence an infinite co-inductive derivation, follows by widening, which was originally conceived [17] as a device for enforcing termination.

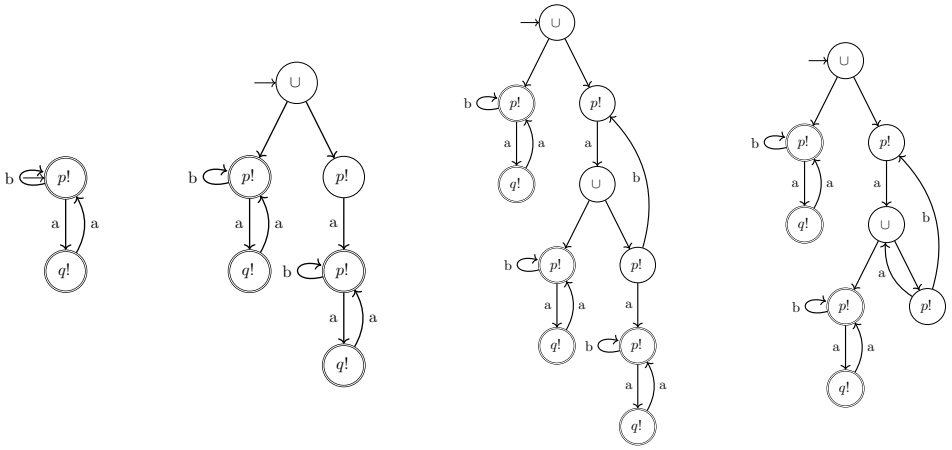


Figure 4 Widened type graphs: (i) $G'_{0,0}$ (ii) $G'_{1,0}$ (iii) $G'_{2,0}$ (iv) $G'_{3,0}$.

5.2 Subtyping algorithm for type graphs

Example 32 illustrates how widening can be applied to establish the existence of an infinite co-inductive proof for $T \leq Q$. Algorithm 1 outlines an algorithm, which applies the same strategy for proving $T \leq T'$, but is applicable when T is not linearly recursive. The key caveat for our algorithm is that T is regular. This ensures $\mathsf{sub}(T)$ is finite. It follows there exists a finite set $wp \subseteq \mathsf{sub}(T)$ such that every cycle of T crosses at least one sub-tree of wp. (wp is inspired by the idea of widening points which are program locations [4] where widening is applied.)

▶ **Example 33.** Continuing Example 32, observe $\mathsf{sub}(T) = \{T, T_1, T_2, T_3\}$ and the only cycle of T crosses each sub-tree of $\mathsf{sub}(T)$. Thus put $wp = \{T_2\}$, though other choices are possible.

In addition to wp, which is fixed throughout, our subtyping algorithm has a worklist L and a partial map $\mathcal{G} : \mathsf{sub}(T) \rightharpoonup \mathbb{G}$ as parameters. $\mathcal{G}$ mirrors the strategy used Example 32 which associates each sub-tree of T with its own type graph. The algorithm is primed with $L = [\langle T, \Gamma \rangle]$ where $\Gamma = \emptyset$ and $\mathcal{G} = \{T \mapsto T'\}$ (recall $\mathbb{T} \subseteq \mathbb{G}$). The algorithm exits successfully at line (2) returning the certificate $\mathcal{G}$, and otherwise throws an exception at (8), (9), (14) or (15) indicating an inconclusive verdict. The body of SUBTYPING removes the first pair $\langle T, \Gamma \rangle$ from the worklist (if not empty), and checks whether the action at the root of T is a receive, a send, or an end. In the latter case, if $\mathbb{G}(T) = \mathsf{end}$ then rule RefEnd of Figure 1 holds and processing continues, otherwise an exception is thrown.

If T is a receive, the progress condition of RefInA is checked at (8). The coverage condition at (9) can be decided without recourse to γ by an auxiliary predicate cover defined thus: $\mathsf{cover}(\emptyset) = \mathit{true}$, $\mathsf{cover}(\cup_{j \in J} G_j) = \wedge_{j \in J} \mathsf{cover}(G_j)$ and $\mathsf{cover}(G) = \exists i \in I : \delta_{p?a_i}(G) \neq \emptyset$ if $G \in \mathbb{T}$. Then $\mathsf{cover}(\mathcal{G}(T))$ holds iff $\forall T' \in \gamma(\mathcal{G}(T)) : \exists i \in I : \delta_{p?a_i}(T') \neq \emptyset$. Line (9) also checks $J \neq \emptyset$. If these checks are passed, $\mathcal{G}$ is relaxed to $\mathcal{G}'$ by updating $\mathcal{G}$ on the keys $\{T_j \mid j \in J\}$. If the sub-tree $T_j \in wp$ then widening is triggered and $\mathcal{G}'(T_j) = \mathcal{G}'(T_j) \triangledown \pi_{p?a_j}(\mathcal{G}(T))$. It follows $\mathcal{G}(T_j) \preceq \mathcal{G}'(T_j)$ and $\pi_{p?a_j}(\mathcal{G}(T)) \preceq \mathcal{G}'(T_j)$, both assuring soundness while ensuring that $\mathcal{G}'(T_j)$ cannot be relaxed ad infinitum. If $T_j \notin wp$ then $\mathcal{G}'(T_j) = \mathcal{G}'(T_j) \cup \pi_{p?a_j}(\mathcal{G}(T))$, again yielding a sound relaxation. Widening at wp is enough to ensure $\mathcal{G}$ is ultimately stable across all keys [4]: thus $\mathcal{G}' = \mathcal{G}$ after a finite number of updates.

To demonstrate RefInA holds it remains to show $\Gamma[T \mapsto \emptyset] + p? \vdash T_j \leq \gamma(\mathcal{G}'(T_j))$ for all $j \in J$. This is achieved by the list comprehension at (11) which extends the worklist

Algorithm 1 Subtyping algorithm where a type graph represents a set of sessions trees.

(1) **function** SUBTYPE(L, wp, $\mathcal{G}$)
(2) **if** $L = \emptyset$ **then** $\mathcal{G}$
(3) **else**
(4) $\langle T, \Gamma \rangle := hd(L)$
(5) **switch** T
(6) **case** $\&_{i \in I}\, p?a_i.T_i$:
(7) $J := \{j \in I \mid \pi_{p?a_j}(\mathcal{G}(T)) \neq \emptyset\}$
(8) **if** $T \mapsto Q \in \Gamma \wedge Q \not\trianglerighteq act(hd(\mathcal{G}(T)))$ **then throw maybe**
(9) **if** $J = \emptyset \vee \exists T' \in \gamma(\mathcal{G}(T)) : \forall i \in I : \delta_{p?a_i}(T') = \emptyset$ **then throw maybe**
(10) $\mathcal{G}' := \mathcal{G}[T_j \mapsto \mathcal{G}(T_j)$ (**if** $T_j \in wp$ **then** $\triangledown$ **else** $\cup$) $\pi_{p?a_j}(\mathcal{G}(T)) \mid j \in J]$
(11) $L' := [\langle T_j, \Gamma[T \mapsto \emptyset] + p? \rangle \mid j \in J, \Gamma(T_j) = \bot \vee \neg(\mathcal{G}'(T_j) \preceq \mathcal{G}(T_j))] :: tl(L)$
(12) SUBTYPE(L', wp, $\mathcal{G}'$)
(13) **case** $\oplus_{i \in I}\, p!a_i.T_i$:
(14) **if** $T \mapsto Q \in \Gamma \wedge Q \not\trianglerighteq act(hd(\mathcal{G}(T)))$ **then throw maybe**
(15) **if** $\exists i \in I : \Pi^k_{p!a_i}(\mathcal{G}(T)) = \emptyset$ **then throw maybe**
(16) $\mathcal{G}' := \mathcal{G}[T_i \mapsto \mathcal{G}(T_j)$ (**if** $T_j \in wp$ **then** $\triangledown$ **else** $\cup$) $\Pi^k_{p!a_i}(\mathcal{G}(T)) \mid i \in I]$
(17) $L' := [\langle T_i, \Gamma[T \mapsto \emptyset] + p! \rangle \mid i \in I, \Gamma(T_i) = \bot \vee \neg(\mathcal{G}'(T_i) \preceq \mathcal{G}(T_i))] :: tl(L)$
(18) SUBTYPE(L', wp, $\mathcal{G}'$)
(19) **case** end :
(20) **if** $\mathcal{G}(T) \neq$ end **then throw maybe**
(21) SUBTYPE($tl(L)$, wp, $\mathcal{G}$)

with the pair $\langle T_j, \Gamma[T \mapsto \emptyset] + p? \rangle$ for each $j \in J$ providing the pair actually needs to be considered, that is, if $\Gamma(T_j) = \bot$ or $\neg(\mathcal{G}'(T_j) \preceq \mathcal{G}(T_j))$. Subtyping then reduces to invoking SUBTYPE(L', wp, $\mathcal{G}'$). Observe $\Gamma(T_j) \neq \bot$ on any subsequent visit to T_j, and eventually $\mathcal{G}'(T_j) \preceq \mathcal{G}(T_j)$ by virtue of widening. Termination thus follows.

The case for send on lines (14)–(18) is arguably simpler since J does not need to be computed and coverage does not need to be checked.

6 Implementation

To assess practicality, we have implemented the Subtype algorithm in Scala 3.1.0 in 5800 LOC. Our implementation makes substantial use of immutable data-structure and parser combinator libraries [28] to reconstruct the type graph operations of [32] and the widening of [43]. As well as the derivatives, our codebase provides a visualiser for type graphs which outputs tikz.

We tested our algorithm on 12 binary and 12 multiparty problems, as detailed in Table 1. The problem trickySub < trickySup is taken from Example 24 whereas all other problems are drawn from (or inspired by) existing literature. The binary benchmarks include 8 which possess "complex accumulation patterns" [6] which is a catch-all term for communication patterns which defy heuristic methods [6]. The remaining 4 problems were chosen by size.

Our algorithm was able to automatically prove subtyping for *all* problems, strictly improving on [3, 6] for the binary problems and [18] for the multiparty problems. Our algorithm was also able to establishing subtyping for *all* the remaining 74 binary problems in the benchmark suite of [6]. The strict improvement over the binary algorithm of [3] occurs because the string widening of [14] deployed in [3], imposes a restriction on the syntactic form of consecutive regular sub-expressions, whereas the particular widening developed for

Table 1 Minimum k of $\Pi^k_{p!a}$ required for successful subtyping, iterations of the algorithm (up), time in ms (expressed as parse time+analysis time) with comparison against related algorithms.

benchmark (binary)	k	up	time	[6]	[3]	[18]	benchmark (multiparty)	k	up	time	[6]	[3]	[18]
ctxta1 < ctxta2 [6]	1	14	1+14	✗	✓	✗	trickySub < trickySup	2	16	1+15	✗	✗	✗
ctxtb1 < ctxtb2 [6]	1	14	1+4	✗	✓	✗	FFT1 < FFT2 [11]	1	15	2+1	✗	✗	✓
14may2 < 14may1 [6]	1	8	1+2	✗	✓	✗	altbit1 < altbit2 [18]	1	4	1+0	✗	✗	✓
badseq1 < badseq2 [6]	1	24	1+16	✗	✓	✗	dbuff1 < dbuff2 [18]	1	5	1+1	✗	✗	✓
march3testa1 < march3testa2 [6]	1	17	1+24	✗	✓	✗	ring1 < ring2 [18]	1	2	1+0	✗	✗	✓
aaaaaab1 < aaaaaab2 [6]	1	9	1+2	✗	✓	✗	rchoice1 < rchoice2 [18]	1	3	1+0	✗	✗	✓
ex1-ok-loop < ex2-ok-loop [6]	1	15	1+34	✗	✓	✗	ex181 < ex182 [27]	1	3	2+0	✗	✗	✓
march3testa1 < march3testb2 [6]	1	28	1+15	✗	✗	✗	ex192 < ex191 [27]	1	9	2+2	✗	✓	✗
twinstar < ex2 [6]	1	12	1+13	✓	✓	✗	ex172 < ex171 [27]	1	7	1+0	✗	✗	✓
decidex1 < decidex2 [6]	1	22	1+33	✓	✓	✗	LSWAq2 < LSWAq1 [36]	1	8	2+0	✗	✗	✓
sub-runningex < sup-runningex [6]	1	12	1+40	✓	✓	✗	LSWAr2 < LSWAr1 [36]	1	5	2+0	✗	✗	✓
september1 < september2 [6]	1	20	1+75	✓	✓	✗	EYTb2 < EYTb1 [23]	1	3	1+0	✗	✗	✓

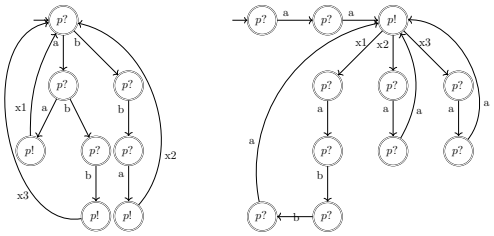


Figure 5 doubleb1 < doubleb2 (adapted from march3testa1 < march3testa2 of [6]).

type graphs [42] used in the present work has no such limitation. However, Figure 5 depicts a binary problem, doubleb1 < doubleb2, delibrately crafted in response to the reviewers, to subvert the type graph widening of [42]. This example shows that the algorithm is incomplete.

Surprisingly, calculating the send derivatives with $k = 1$ was sufficient for all but one problem, trickySub < trickySup, which was crafted to be difficult. The *up* column records how many times the map $\mathcal{G}$ is updated, which governs complexity. Timings are presented as parse+analysis time and were taken on a 32 GB laptop equipped with a 2.8GHz i7 processor. The zero times are runs beneath the granularity of the clock. The slowest runtime was 1+75=76ms, whereas the subtyping tool of [3], takes 1,757ms worse-case for the binary benchmarks (on the same machine), which suggests that performance-wise, multiparty subtyping is no worse than binary subtyping if carefully designed. All certificates are available at `https://www.cs.kent.ac.uk/people/staff/amk/certificates.zip`.

7 Related Work

The undecidability of asynchronous subtyping [7, 35] has given rise to a rich line of research aimed at finding sound algorithms. The first work [8] to explore the boundaries of undecidability, showed that synchronous subtyping can be generalised to k-bounded asynchronous subtyping [8], where input accumulation depth is bounded by k ($k = 0$ is synchronous subtyping). With k fixed, k-bounded asynchronous subtyping is sound and decidable (though the existence of such a k is undecidable [8]).

Asynchronous (binary) approaches include the seminal algorithm from [5, 6] for binary sessions and [3], which uses word approximation [14] to enlarge the class of problems for which asynchronous subtyping can be algorithmically established. The extension from the

binary setting to multiparty is not obvious. For example, swapping two send actions or two receive actions (to encode SW2) is not safe in the binary setting: it violates the FIFO ordering of messages, potentially causing communication mismatches (e.g., $s!\mathrm{a}.\, s!\mathrm{b} \not\leq s!\mathrm{b}.\, s!\mathrm{a}$) or deadlocks (e.g., $c?\mathrm{a}.\, c?\mathrm{b} \not\leq c?\mathrm{b}.\, c?\mathrm{a}$). The multiparty setting adds more interleaving options to handle and requires more machinery to keep track of the actions-role association.

As to multiparty asynchronous subtyping, the algorithm in [18], while applicable to several realistic protocols, does not address "complex accumulation patterns" [6] that can be, instead, handled by the tools of [3, 6]. MAS [27] has been mechanized [23] providing a sound and complete tool for verifying subtyping *given a hand-crafted certificate as input*, but provides no solution to the fundamental problem of how to compute a certificate.

Interest in subtyping seems to be increasing rather than abating and another approach to asynchronous multiparty subtyping [36], based on projecting global types (a global model of a collection of local types), was proposed in parallel to [3] at a sister conference. The global types in [36] provide richer select and branching primitives with respect to classic global types: in a given state, a sender can select to send a message to different receivers, and likewise a receives can wait on different senders. A central requirement of multiparty subtyping in [36] is sub-protocol fidelity [36, Definition 5.1(i)] which ensures that any refinement (subtype) is consistent with a given global type. Sub-protocol fidelity yields a very different notion of subtyping to the one adopted here (and [5, 6, 12, 13] and formalised as MAS [27]). Sub-protocol fidelity allows some actions to be swapped, but it does not allow the anticipation of outputs before inputs, which is a requirement for performance optimisation [38]. This means that a session type may be a subtype of another according to our algorithm (and MAS), but not that of [36]. Thus it is no paradox when the authors of [36] claim decidability while pointing out that asynchronous subtyping is undecidable [7, 35]. Classically, the subtyping question is answered pairwise: by comparing a type against a candidate subtype, without reference to a global type. This formulation is favourable when the global type is unknown, for instance, if types are mined from source code.

Session subtyping is related to compliance between session contracts [1], which is inspired by testing-preorders [19]. Must-testing, in the binary and synchronous setting, can be seen as a sort of inductive compliance relation [39]. Multiparty synchronous must-testing has been studied in [20], binary asynchronous in [2]. Another interesting preorder is fair subtyping [9], which takes its cues from should testing preorder [41], and preserves the possibility of correct termination. Difference nuances of session subtyping, such as those mentioned above, suggest the need of generic subtyping algorithms, parameterised by the underlying preorder.

8 Conclusion

The application of abstract interpretation to asynchronous session subtyping is challenging in the multiparty setting. To provide a semantic basis for abstraction, an alternative notion of SISO-tree refinement is given, and as well as new notions of derivative which, together, enables the MAS subtyping relation to be re-expressed over sets of session-trees. This provides a foundation for abstract interpretation, akin to a classical collecting semantics, whose use is illustrated with a new abstract domain for session trees, constructed from type graphs, that were originally proposed for type recovery. Overall, we provide the first algorithmic formulation of MAS together with experimental results which are truly encouraging.

References

1 Franco Barbanera and Ugo de'Liguoro. Sub-behaviour relations for session-based client/server systems. *Mathematical Structures in Computer Science*, 25(6):1339–1381, 2015. doi:10.1017/S096012951400005X.

2 Giovanni Bernardi, Ilaria Castellani, Paul Laforgue, and Léo Stefanesco. Constructive Characterisations of the MUST-preorder for Asynchrony. In *European Symposium on Programming*, volume 15694 of *Lecture Notes in Computer Science*, pages 88–116, 2025. doi:10.1007/978-3-031-91118-7_4.

3 Laura Bocchi, Andy King, and Maurizio Murgia. Asynchronous Subtyping by Trace Relaxation. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 14570 of *Lecture Notes in Computer Science*, pages 207–226, Berlin, 2024. Springer-Verlag. doi:10.1007/978-3-031-57246-3_12.

4 François Bourdoncle. Efficient Chaotic Iteration Strategies with Widenings. In *Formal Methods in Programming and Their Applications*, volume 735 of *Lecture Notes in Computer Science*, pages 128–141, New York, 1993. Springer-Verlag. doi:10.1007/BFb0039704.

5 Mario Bravetti, Marco Carbone, Julien Lange, Nobuko Yoshida, and Gianluigi Zavattaro. A Sound Algorithm for Asychronous Session Subtyping. In *International Conference on Concurrency Theory*, volume 140 of *LIPIcs*, pages 38:1–38:16, Dagstuhl, 2019. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CONCUR.2019.38.

6 Mario Bravetti, Marco Carbone, Julien Lange, Nobuko Yoshida, and Gianluigi Zavattaro. A Sound Algorithm for Asychronous Session Subtyping and its Implementation. *Logical Methods in Computer Science*, 17(1):1–35, 2021. doi:10.23638/LMCS-17(1:20)2021.

7 Mario Bravetti, Marco Carbone, and Gianluigi Zavattaro. Undecidability of Asynchronous Session Subtyping. *Information and Computation*, 256:300–320, 2017. doi:10.1016/j.ic.2017.07.010.

8 Mario Bravetti, Marco Carbone, and Gianluigi Zavattaro. On the Boundary between Decidability and Undecidability of Asynchronous Session Subtyping. *Theoretical Computer Science*, 722:19–51, 2018. doi:10.1016/j.tcs.2018.02.010.

9 Mario Bravetti, Julien Lange, and Gianluigi Zavattaro. Fair Asynchronous Session Subtyping. *Logical Methods in Computer Science*, 20:5:1–5:47, 2024. doi:10.48550/arXiv.2101.08181.

10 Janusz Brzozowski. Derivatives of Regular Expressions. *Journal of the ACM*, 11(4):481–494, 1964. doi:10.1145/321239.321249.

11 David Castro-Perez and Nobuko Yoshida. CAMP: cost-aware multiparty session protocols. *Proc. ACM Program. Lang.*, 4(OOPSLA):155:1–155:30, 2020. doi:10.1145/3428223.

12 Tzu-Chun Chen, Mariangiola Dezani-Ciancaglini, Alceste Scalas, and Nobuko Yoshida. On the Preciseness of Subtyping in Session Types. *Logical Methods in Computer Science*, 13(2):1–61, 2017. doi:10.23638/LMCS-13(2:12)2017.

13 Tzu-Chun Chen, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. On the Preciseness of Subtyping in Session Types. In *Principles and Practice of Declarative Programming*, pages 135–146, New York, 2014. ACM Press. doi:10.1145/2643135.2643138.

14 Tae-Hyoung Choi, Oukseh Lee, Hyunha Kim, and Kyung-Goo Doh. A Practical String Analyzer by the Widening Approach. In *Asian Symposium on Programming and Systems*, volume 4279 of *Lecture Notes in Computer Science*, pages 374–388, Berlin, 2006. Springer-Verlag. doi:10.1007/11924661_23.

15 Liron Cohen. Non-well-founded Deduction for Induction and Coinduction. In *Conference on Automated Deduction*, volume 12699 of *Lecture Notes in Computer Science*, pages 3–24. Springer-Verlag, 2021. doi:10.1007/978-3-030-79876-5_1.

16 Bruno Courcelle. Fundamental Properties of Infinite Trees. *Theoretical Computer Science*, 25:95–169, 1983. doi:10.1016/0304-3975(83)90059-2.

17 Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Principles of*

Programming Languages, pages 238–252, New York, 1977. ACM Press. doi:10.1145/512950.512973.

18 Zak Cutner, Nobuko Yoshida, and Martin Vassor. Deadlock-Free Asynchronous Message Reordering in Rust with Multiparty Session Types. In *Symposium on Principles and Practice of Parallel Programming*, pages 246–261, New York, 2022. ACM Press. doi:10.1145/3503221.3508404.

19 Rocco De Nicola and Matthew Hennessy. Testing Equivalences for Processes. *Theoretical Computer Science*, 34:83–133, 1984. doi:10.1007/BFb0036936.

20 Rocco De Nicola and Hernán Melgratti. Multiparty Testing Preorders. *Logical Methods in Computer Science*, 19:1:1–1:31, 2016. doi:10.46298/lmcs-19(1:1)2023.

21 Romain Demangeon and Kohei Honda. Full Abstraction in a Subtyped pi-Calculus with Linear Types. In *International Conference on Concurrency Theory*, volume 6901 of *Lecture Notes in Computer Science*, pages 280–296, Berlin, 2011. Springer-Verlag. doi:10.1007/978-3-642-23217-6_19.

22 Romain Demangeon, Kohei Honda, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. Practical interruptible conversations: distributed dynamic verification with multiparty session types and Python. *Formal Methods in Systems Design*, 46(3):197–225, 2015. doi:10.1007/s10703-014-0218-8.

23 Burak Ekici and Nobuko Yoshida. Completeness of asynchronous session tree subtyping in coq. In Yves Bertot, Temur Kutsia, and Michael Norrish, editors, *International Conference on Interactive Theorem Proving*, volume 309 of *LIPIcs*, pages 13:1–13:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPICS.ITP.2024.13.

24 Simon Fowler. An Erlang Implementation of Multiparty Session Actors. In *Interaction and Concurrency Experience*, volume 223 of *EPTCS*, pages 36–50, 2016. doi:10.4204/EPTCS.223.3.

25 Simon Gay and Malcolm Hole. Types and Subtypes for Client-Server Interactions. In *European Symposium on Programming*, volume 1576 of *Lecture Notes in Computer Science*, pages 74–90, Berlin, 1999. Springer-Verlag. doi:10.1007/3-540-49099-X_6.

26 Simon Gay and Malcolm Hole. Subtyping for Session Types in the Pi Calculus. *Acta Informatica*, 42:191–225, 2005. doi:10.1007/s00236-005-0177-z.

27 Silvia Ghilezan, Jovanka Pantovic, Ivan Prokic, Alceste Scalas, and Nobuko Yoshida. Precise Subtyping for Asynchronous Multiparty Sessions. *Proc. ACM Program. Lang.*, 5(POPL):1–28, 2021. doi:10.1145/3434297.

28 Steve Hill. Combinators for Parsing Expressions. *Journal of Functional Programming*, 6(3):445–463, 1996. doi:10.1017/S0956796800001799.

29 Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language Primitives and Type Disciplines for Structured Communication-based Programming. In *European Symposium on Programming*, volume 1381 of *Lecture Notes in Computer Science*, pages 22–138, Berlin, 1998. Springer-Verlag. doi:10.1007/BFb0053567.

30 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *Principles of Programming Languages*, pages 273–284. ACM, 2008. doi:10.1145/1328438.1328472.

31 Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-Based Distributed Programming in Java. In *European Conference on Object-Oriented Programming*, volume 5142 of *Lecture Notes in Computer Science*, pages 516–541, Berlin, 2008. Springer-Verlag. doi:10.1007/978-3-540-70592-5_22.

32 Gerda Janssens and Maurice Bruynooghe. Deriving Descriptions of Possible Values of Program Variables by Means of Abstract Interpretation. *Journal of Logic Programming*, 13(2&3):205–258, 1992. doi:10.1016/0743-1066(92)90032-X.

33 Nicolas Lagaillardie, Rumyana Neykova, and Nobuko Yoshida. Stay Safe Under Panic: Affine Rust Programming with Multiparty Session Types. In *European Conference on Object-Oriented*

Programming, volume 222, pages 4:1–4:29, Dagstuhl, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ECOOP.2022.4.

34 Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. Fencing off Go: liveness and safety for channel-based programming. In *Principles of Programming Languages*, pages 748–761, 2017. doi:10.1145/3009837.3009847.

35 Julien Lange and Nobuko Yoshida. On the Undecidability of Asynchronous Session Subtyping. In *Foundations of Software Science and Computation Structures*, volume 10203 of *Lecture Notes in Computer Science*, pages 441–457, Berlin, 2017. Springer-Verlag. doi:10.1007/978-3-662-54458-7_26.

36 Elaine Li, Felix Stutz, and Thomas Wies. Deciding Subtyping for Asynchronous Multiparty Sessions. In *European Symposium on Programming*, volume 14576 of *Lecture Notes in Computer Science*, pages 176–205, Berlin, 2024. Springer-Verlag. doi:10.1007/978-3-031-57262-3_8.

37 Dimitris Mostrous and Vasco Vasconcelos. Session Typing for a Featherweight Erlang. In *Coordination Models and Languages*, Lecture Notes in Computer Science, pages 95–109, 2011. doi:10.1007/978-3-642-21464-6_7.

38 Dimitris Mostrous, Nobuko Yoshida, and Kohei Honda. Global Principal Typing in Partially Commutative Asynchronous Sessions. In *European Symposium on Programming*, volume 5502 of *Lecture Notes in Computer Science*, pages 316–332, Berlin, 2009. Springer-Verlag. doi:10.1007/978-3-642-00590-9_23.

39 Maurizio Murgia. A fixed-points based framework for compliance of behavioural contracts. *J. Log. Algebraic Methods Program.*, 120:100641, 2021. doi:10.1016/j.jlamp.2021.100641.

40 Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

41 Arend Rensink and Walter Vogler. Fair testing. *Information and Computation*, 205(2):125–198, 2007. doi:10.1016/j.ic.2006.06.002.

42 Pascal Van Hentenryck, Agostino Cortesi, and Baudouin Le Charlier. Type Analysis of Prolog Using Type Graphs. In *Programming Language Design and Implementation*, pages 337–348. ACM Press, 1994. doi:10.1145/773473.178479.

43 Pascal Van Hentenryck, Agostino Cortesi, and Baudouin Le Charlier. Type Analysis of Prolog Using Type Graphs. *Journal of Logic Programming*, 22(3):179–209, 1995. doi:10.1016/0743-1066(94)00021-W.

A Sound and Complete Characterization of Fair Asynchronous Session Subtyping

Mario Bravetti
University of Bologna, Italy

Luca Padovani
University of Bologna, Italy

Gianluigi Zavattaro
University of Bologna, Italy

Abstract

Session types are abstractions of communication protocols enabling the static analysis of message-passing processes. Refinement notions for session types are key to support safe forms of process substitution while preserving their compatibility with the rest of the system. Recently, a fair refinement relation for asynchronous session types has been defined allowing the anticipation of message outputs with respect to an unbounded number of message inputs. This refinement is useful to capture common patterns in communication protocols that take advantage of asynchrony. However, while the semantic (*à la testing*) definition of such refinement is straightforward, its characterization has proved to be quite challenging. In fact, only a sound but not complete characterization is known so far. In this paper we close this open problem by presenting a sound and complete characterization of asynchronous fair refinement for session types. We relate this characterization to those given in the literature for *synchronous* session types by leveraging a novel labelled transition system of session types that embeds their asynchronous semantics.

2012 ACM Subject Classification Theory of computation → Type theory

Keywords and phrases Binary sessions, session types, fair asynchronous subtyping

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2025.11

Related Version *Full Version*: `https://arxiv.org/abs/2506.06078` [7]

Funding The authors have been partially supported by the ANR project SmartCloud ANR-23-CE25-0012 and the PNRR project CN–HPC Spoke 9 Innovation Grant RTMER.

Acknowledgements We are grateful to the anonymous CONCUR'25 reviewers for their feedback.

1 Introduction

Abstract models such as communicating finite-state machines [3] and asynchronous session types [18] are essential to reason about correctness of distributed systems whose components communicate with point-to-point FIFO channels. A fundamental issue, which makes it possible to manage system correctness in a compositional way, is the development of techniques allowing a component to be *refined* independently of the others, without compromising the correctness of the whole system. In this respect, the notion of fair asynchronous refinement for session types introduced by Bravetti, Lange and Zavattaro [6] guarantees fair termination, *i.e.* successful termination under fairness assumption: execution traces that loop forever are deemed unrealistic/unimportant and are not considered. Such a notion implies all the desirable safety and liveness properties of communicating systems, including communication safety, deadlock freedom, absence of orphan messages, and lock freedom. However, while the semantic (*à la testing*) definition of such refinement is straightforward, its (coinductive) characterization (*à la session subtyping*) has proved to be quite challenging. In fact, Bravetti, Lange and Zavattaro [6] only provide a sound but not complete characterization.

© Mario Bravetti, Luca Padovani, and Gianluigi Zavattaro;
licensed under Creative Commons License CC-BY 4.0

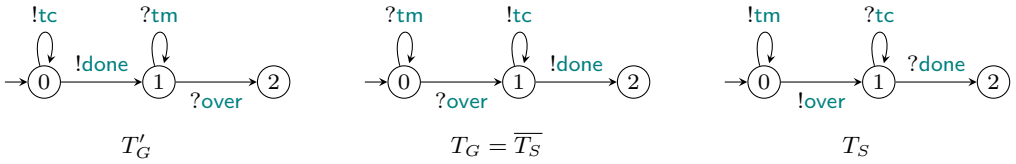


Figure 1 Satellite protocols. T'_G is the refined session type of the ground station, T_G is the session type of ground station, and T_S is the session type of the spacecraft.

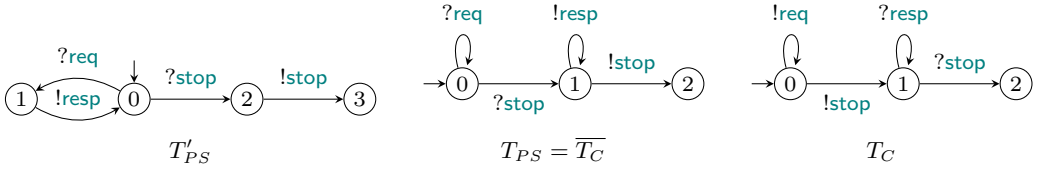


Figure 2 Stream processing server T'_{PS} is the refined session type of the batch processing server T_{PS}, and T_C is the session type of the client.

With respect to previous notions of asynchronous session subtyping [22, 9] (which leverage asynchrony by allowing the refined component to anticipate message emissions, but only under certain conditions) fair asynchronous subtyping [6] makes it possible to encompass subtypes that occur naturally in communication protocols, *e.g.* where two parties simultaneously send each other a finite but unspecified amount of messages before consuming them from their buffers. We illustrate this scenario in Figure 1, which depicts the interaction between a spacecraft S and a ground station G that communicate via two unbounded asynchronous channels (one in each direction). For convenience, the protocols are represented as communicating finite-state machines [3], *i.e.* with a labeled transition system-like notation where "?" represents inputs, "!" represents outputs, and the initial state is indicated by an incoming arrow. Consider T_S and T_G first. Session type T_S is the abstraction of the spacecraft, which may send a finite but unspecified number of telemetries tm (abstractly modeling a `for` loop), followed by a message over. In the second phase, the spacecraft receives a number of telecommands (tc), followed by a message done. In principle, the ground station should behave as the type T_G which is the dual of T_S (denoted by $\overline{T_S}$) where outputs become inputs and vice-versa. However, since the flyby window may be short, it makes sense to implement the ground station so that it communicates with the satellite by anticipating the output of the commands. That is, the ground station follows the type T'_G, which sends telecommands before receiving telemetries. In this way T'_G and T_S interact in a symmetric manner: they first send all of their messages and then consume the messages sent from the other partner. No communication error can occur, and the communication protocol can always terminate successfully, with empty queues. In fact T'_G and T_S also form a correct composition in the sense that every message that is sent by one participant is eventually received by the other. The session subtyping presented by Bravetti, Lange and Zavattaro [6], which is proved to be a sound characterization of fair asynchronous session refiment, makes it possible to actually *prove* that T'_G refines T_G.

Let us now consider, in Figure 2, a more complex scenario concerning a processing server PS and its client C. Consider T_{PS} and T_C first. Session type T_{PS} is the abstraction of a batch processing server and T_C that of a client. Note that these types are respectively isomorphic to T_G and T_S of Figure 1: now the client T_C sends generic requests req (instead of specific telemetry data tm) and, when it decides to stop sending via stop, it keeps waiting

for responses resp (instead of telecommand data tc) until it receives stop. As in the previous scenario, the client T_C and the batch processing server T_{PS} (which is its dual) obviously form a correct composition. Also here it is possible to consider a more efficient version of the processing server, *i.e.* a stream processing server T'_{PS} which immediately (and asynchronously) sends the response to each request it receives. In this scenario, it may be natural to send responses one at a time after each request. In fact also T'_{PS} and T_C form a correct composition. Actually, we have that T'_{PS} is a fair asynchronous session refinement of T_{PS}, meaning that T'_{PS} is a good replacement of T_{PS} in any possible context. However, the characterization provided by Bravetti, Lange and Zavattaro [6] is unable to capture this specific refinement.

Technically speaking, the coinductive characterization of subtyping given by Bravetti, Lange and Zavattaro [6] is not complete because it does not support *output covariance*, that is the possibility for a subtype to remove branches in output choices, corresponding to a reduced internal nondeterminism. This feature is needed to relate T'_{PS} and T_{PS} of Figure 2 because the state 1 (resp. 2) of T'_{PS}, in which the forced internal choice of emitting !resp (resp. !stop) is taken, has a unique corresponding state in T_{PS}, namely state 1, where two possible choices are present (either !resp or !stop can be sent).

In this paper we close the open problem of finding a sound and complete characterization of asynchronous fair refinement for session types. The characterization we present is crucially based on a novel asynchronous semantics for session types that: (i) expresses their asynchronous behavior without explicit modeling of FIFO buffers (as in the original paper of Bravetti, Lange and Zavattaro [6]), and (ii) allows us to adapt, to the asynchronous case, the approach introduced by Ciccone and Padovani [23, 12] for providing complete characterizations of *fair session subtyping* in the *synchronous* case. Being complete, our new subtyping allows us to prove that the stream processing server T'_{PS} is a refinement of the batch processing server T_{PS}.

Structure of the paper. In Section 2 we recall the notion of fair asynchronous refinement and the buffer-based asynchronous semantics of session types given by Bravetti, Lange and Zavattaro [6]. In Section 3 we present our novel alternative asynchronous semantics and prove that it is equivalent to the previous one, in the sense that it characterizes the same notion of session composition correctness. In Section 4 we present a sound and complete characterization of asynchronous fair refinement based on the semantics of Section 3. We conclude in Section 5 with a more detailed description of related work and hints at further developments. Proofs and additional technical material can be found in the long version of the paper [7].

2 Preliminaries

We start by recalling the syntax and asynchronous semantics of session types as well as the notion of fair refinement given by Bravetti, Lange and Zavattaro [6]. In the definition of the syntax, instead of using the $\oplus$, &, and the *rec* operators, we consider an equivalent process algebraic notation with choice and prefix [20] and a coinductive syntax to deal with possibly infinite session types. Following Bravetti, Lange and Zavattaro [6] we restrict to first-order session types, *i.e.* session types in which send/receive operations are used to exchange only messages of elementary *singleton type*. Considering higher-order types, where send/receive operations can be used to exchange also sessions, would not modify significantly the proofs of our results.

We assume the existence of a set of *singleton types* ranged over by a, b, ... that we use to label branches in session types. The only value of type a is called *tag* and is denoted by the same symbol a. *Pre-session types* are the possibly infinite trees coinductively generated by the productions below:

$$\textbf{Polarity} \quad p ::= ? \mid !$$
$$\textbf{Pre-session type} \quad S, T ::= \mathsf{end} \mid \textstyle\sum_{i \in I} p\mathsf{a}_i.S_i$$

The *polarities* ? and ! are used to distinguish *input actions* from *output actions*. A *session type* is a pre-session type that satisfies the following well-formedness conditions:

1. in every subtree of the form $\sum_{i \in I} p\mathsf{a}_i.S_i$, I is a non-empty finite set and if I contains more than one element, then the message types a_i are pairwise distinct tags;
2. the tree is *regular*, namely it is made of finitely many distinct subtrees $\sum_{i \in I} p\mathsf{a}_i.S_i$. Courcelle [13] established that every such tree can be expressed as the unique solution of a finite system of equations $\{S_i = T_i\}_{i \in I}$ where each metavariable S_i may occur (guarded) in the T_j's;
3. every subtree $\sum_{i \in I} p\mathsf{a}_i.S_i$ contains a leaf of the form end.

A session type end describes a terminated session (*i.e.* the corresponding channel is no longer usable). A branching session type $\sum_{i \in I} p\mathsf{a}_i.S_i$ describes a channel used for sending or receiving a message of type a_i and then according to S_i. The well-formedness condition (1) above requires that message types must be pairwise distinct tags if there is more than one branch. Sometimes we write $p\mathsf{a}_1.S_1 + \cdots + p\mathsf{a}_n.S_n$ for $\sum_{i=1}^{n} p\mathsf{a}_i.S_i$. Note that the polarity p is the same in every branch, *i.e.* mixed choices are disallowed as usual.

The well-formedness condition (2) guarantees that we deal only with session types representing trees consisting of finitely many distinct subtrees as those that can be represented with a finite syntax and the *rec* operator [13].

The well-formedness condition (3) ensures that session types describe protocols that can always terminate. This assumption is not usually present in other session type syntax definitions where it is possible to also express non-terminating protocols. As we will see in the following (Definition 3), non-terminating protocols are not inhabited in our theory. In particular, we consider a notion of correct type composition such that, whatever sequence of interactions is performed, such sequence can always be extended to reach successful session termination where both types become end. In conclusion, ruling out non-terminating protocols in the syntax of session types does not affect the family of protocols we are interested in modeling and at the same time simplifies the technical development.

It is worth to mention that Bravetti, Lange and Zavattaro [6] do not consider the well formedness condition (3), hence allowing for the specification of non-terminating types. In the comparison with the related literature in Section 5 we discuss the impact of the absence of this condition on the definition of the fair asynchronous subtyping reported in that paper.

We write $\overline{S}$ for the *dual* of S, which is the session type corecursively defined by the equations

$$\overline{\mathsf{end}} = \mathsf{end} \qquad \overline{\textstyle\sum_{i \in I} p\mathsf{a}_i.S_i} = \textstyle\sum_{i \in I} \overline{p}\mathsf{a}_i.\overline{S_i}$$

where $\overline{p}$ is the dual of the polarity p so that $\overline{?} = !$ and $\overline{!} = ?$. As usual, duality affects the direction of messages but not message types.

We now define the transition system that we use to formalize the notion of correct session type composition and the induced notion of refinement [6].

The transition system makes use of explicit FIFO queues of messages that have been sent asynchronously. A *configuration* is a term $[S, \omega_S]|[T, \omega_T]$ where S and T are session types equipped with two queues ω_S and ω_T of incoming messages. We write ϵ for the empty queue and we use s, s', etc. to range over configurations.

▶ **Definition 1** (Transition Relation [6]). *The transition relation $\rightarrow$ over configurations is the minimal relation satisfying the rules below (plus symmetric ones, omitted):*
1. *if $j \in I$ then $[\sum_{i\in I} !\mathsf{a}_i.S_i, \omega_S]||[T, \omega_T] \rightarrow [S_j, \omega_S]||[T, \omega_T \mathsf{a}_j]$;*
2. *if $j \in I$ then $[\sum_{i\in I} ?\mathsf{a}_i.S_i, \mathsf{a}_j\omega_S]||[T, \omega_T] \rightarrow [S_j, \omega_S]||[T, \omega_T]$.*

We write $\rightarrow^*$ for the reflexive and transitive closure of the $\rightarrow$ relation.

▶ **Example 2.** Consider types $S = !\mathsf{a}.?\mathsf{a}.\mathsf{end} + !\mathsf{b}.?\mathsf{b}.\mathsf{end}$ and $T = !\mathsf{a}.?\mathsf{a}.\mathsf{end}$. The configuration $[S, \epsilon]||[T, \epsilon]$ has the sequence of transitions $[S, \epsilon]||[T, \epsilon] \rightarrow [?\mathsf{a}.\mathsf{end}, \epsilon]||[T, \mathsf{a}] \rightarrow [?\mathsf{a}.\mathsf{end}, \mathsf{a}]||[?\mathsf{a}.\mathsf{end}, \mathsf{a}] \rightarrow [?\mathsf{a}.\mathsf{end}, \mathsf{a}]||[\mathsf{end}, \epsilon] \rightarrow [\mathsf{end}, \epsilon]||[\mathsf{end}, \epsilon]$, which terminates in a configuration with both types equal to end and with both message queues empty. It also has, e.g., another computation $[S, \epsilon]||[T, \epsilon] \rightarrow [S, \mathsf{a}]||[?\mathsf{a}.\mathsf{end}, \epsilon] \rightarrow [?\mathsf{b}.\mathsf{end}, \mathsf{a}]||[?\mathsf{a}.\mathsf{end}, \mathsf{b}]$ which terminates in a configuration with both types different from end and with both message queues not empty. ⌟

Intuitively, the composition of the two types S and T in Example 2 is *incorrect* because it can lead to a deadlocked configuration $[?\mathsf{b}.\mathsf{end}, \mathsf{a}]||[?\mathsf{a}.\mathsf{end}, \mathsf{b}]$ discussed therein. Following Bravetti, Lange and Zavattaro [6] we formalize the correct composition of two types as a *compliance* relation.

▶ **Definition 3** (Compliance [6]). *Given a configuration s we say that it is a correct composition if, whenever $s \rightarrow^* s'$, then also $s' \rightarrow^* [\mathsf{end}, \epsilon]||[\mathsf{end}, \epsilon]$.*
Two session types S and T are compliant *if $[S, \epsilon]||[T, \epsilon]$ is a correct composition.*

We can now formally state that the types S and T of Example 2 are not *compliant* because of the sequence of transitions discussed at the end of Example 2: $[S, \epsilon]||[T, \epsilon] \rightarrow [S, \mathsf{a}]||[?\mathsf{a}.\mathsf{end}, \epsilon] \rightarrow [?\mathsf{b}.\mathsf{end}, \mathsf{a}]||[?\mathsf{a}.\mathsf{end}, \mathsf{b}]$ leading to a configuration different from the successfully terminated computation $[\mathsf{end}, \epsilon]||[\mathsf{end}, \epsilon]$, and without outgoing transitions.

Compliance induces a semantic notion of type refinement, as follows.

▶ **Definition 4** (Refinement [6]). *A session type S refines T, written $S \preceq T$, if, for every R s.t. T and R are compliant, then S and R are also compliant.*

In words, this definition says that a process behaving as T can be "safely" replaced by a process behaving as S when S is a refinement of T. Indeed, the peer process, which is assumed to behave according to some session type R such that R and T are compliant, will still interact correctly after the substitution has taken place. We quote "safely" above since we want to stress that the notion of session correctness being preserved here is not merely a *safety property*, but rather a combination of a safety property (if the interaction gets stuck, then it has successfully terminated) and a liveness property (the interaction can always successfully terminate).

Because of the universal quantification on the type R, Definition 4 provides few insights about what it means for S to be a refinement of T. For this reason, it is important to provide alternative (but equivalent) characterizations of type refinement. Bravetti, Lange and Zavattaro [6] present a characterization of refinement defined coinductively *à la session subtyping* [16], hence called fair asynchronous subtyping, which can be used to prove some interesting and non trivial relations as those discussed in Section 1.

▶ **Example 5** (Ground station and satellite communication [6]). We now present the session types describing the possible behaviours of the ground station in the satelllite communication example presented in Section 1. We use type T to represent the communication behaviour of the automaton T_G in Figure 1: $T = ?\mathsf{tm}.T + ?\mathsf{over}.T'$ where $T' = !\mathsf{tc}.T' + !\mathsf{done}.\mathsf{end}$. We use S for the automaton T'_G: $S = !\mathsf{tc}.S + !\mathsf{done}.S'$ where $S' = ?\mathsf{tm}.S' + ?\mathsf{over}.\mathsf{end}$. Type T specifies the initially expected behaviour of the ground station as a complementary protocol

w.r.t. to the one, T_S, followed by the spacecraft. Type S specifies an implementation where the emission of the telecommands is anticipated w.r.t. the consumption of the received telemetries. The subtyping of [6] allows to prove that $S \preceq T$, hence we can conclude that the protocol with the anticipation of the telemetries correctly implements (refines) the expected communication behaviour of the ground station. ⌟

The coinductive subtyping in [6] is a sound but not complete characterization of fair asynchronous refinement. In particular, it is uncapable of establishing relations $S \preceq T$ where S describes a more deterministic protocol than T (output covariance). We have seen an instance of this case in Section 1 when discussing the stream and batch processing servers. Liveness-preserving refinements are difficult to characterize because they are intrinsically non-local: the removal of an output action in a region of a session type may compromise the successful termination of the protocol along a branch that is "far away" from the location where the output was removed. In our case this phenomenon is further aggravated by the presence of asynchrony, which allows output actions to be anticipated and therefore to be moved around.

In order to provide a complete characterization of refinement, we use an alternative, queue-less semantics of asynchronous session types that we introduce in the next section.

3 An Alternative Asynchronous Semantics for Session Types

The first step towards our characterization of fair refinement is the definition of a novel asynchronous semantics for session types that does not make use of explicit queues. This alternative semantics will allow us to adapt the complete characterization of fair synchronous subtyping for session types [23, 12] to the asynchronous case.

We now introduce a *labelled transition system* to describe the sequences of input/output actions that are allowed by a session type. A *label* is a pair made of a *polarity* and a *tag* a:

Label $\quad \alpha, \beta ::= p\mathsf{a}$

We will refer to the dual of a label α with the notation $\overline{\alpha}$, where $\overline{p\mathsf{a}} \stackrel{\text{def}}{=} \overline{p}\mathsf{a}$.

The labelled transition system (LTS) of asychronous session types is *coinductively* defined thus:

$$\text{[L-SYNC]}\quad \frac{}{\sum_{i\in I} p\mathsf{a}_i.S_i \xrightarrow{p\mathsf{a}_k} S_k}\ k \in I \qquad \text{[L-ASYNC]}\quad \frac{\forall i \in I : S_i \xrightarrow{?\mathsf{a}} T_i}{\sum_{i\in I} !\mathsf{a}_i.S_i \xrightarrow{?\mathsf{a}} \sum_{i\in I} !\mathsf{a}_i.T_i}$$

The rule [L-SYNC] simply expresses the fact that a session type allows the input/output action described by its topmost prefix, as expected. The rule [L-ASYNC] is concerned with asynchrony and lifts input actions that are enabled deep in a session type, provided that such actions are only prefixed by output actions. The rationale for such "asynchronous transitions" is that the topmost outputs allowed by a session type may have been performed asynchronously, so that the underlying input is enabled even if the output messages have not been consumed yet. For example, the session type $!\mathsf{a}.?\mathsf{b}.S$ may evolve either as

$$!\mathsf{a}.?\mathsf{b}.S \xrightarrow{!\mathsf{a}} ?\mathsf{b}.S \xrightarrow{?\mathsf{b}} S \qquad \text{or as} \qquad !\mathsf{a}.?\mathsf{b}.S \xrightarrow{?\mathsf{b}} !\mathsf{a}.S \xrightarrow{!\mathsf{a}} S \tag{1}$$

The first transition sequence is the expected one and describes an evolution in which the order of actions is consistent with the syntactic structure of the session type. The second

transition sequence is peculiar to the asynchronous setting and describes an evolution in which the input ?b *is performed before the output* !a. Asynchronous transitions seem to clash with the very nature of session types, whose purpose is to impose an order on the actions performed by a process on a channel. To better understand them, we find it useful to appeal to the usual interpretation of labelled transition systems as descriptions of those actions that trigger interactions with the environment: a process that behaves according to the session type !a.?b.S is still required to send a before it starts waiting for b. However, it will be capable of receiving b before a has been consumed by the party with which it is interacting, because a may have been queued. In this sense, we see the queue as an entity associated with the process and not really part of the environment in which the process executes.

It is useful to introduce some additional notation related to actions and transitions of session types. We let φ and ψ range over strings of labels and use ε to denote the empty string. We write $\xrightarrow{\alpha_1 \cdots \alpha_n}$ for the composition $\xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n}$, we write $S \xrightarrow{\varphi}$ if $S \xrightarrow{\varphi} T$ for some T and we write $S \not\xrightarrow{\varphi}$ if not $S \xrightarrow{\varphi}$.

Two subtle aspects of the LTS deserve further comments. First of all, asynchronous transitions are possible only provided that they are enabled *along every branch* of a session type. For example, if $S = \mathsf{!a}.\mathsf{?c}.S_1 + \mathsf{!b}.(\mathsf{?c}.S_2 + \mathsf{?d}.S_3)$, then we have

$$S \xrightarrow{\mathsf{?c}} \mathsf{!a}.S_1 + \mathsf{!b}.S_2 \qquad \text{and} \qquad S \not\xrightarrow{\mathsf{?d}} \qquad \text{and} \qquad S \xrightarrow{\mathsf{!b?d}} S_3$$

The ?c-labelled transition is enabled because that input action is available regardless of the message (either a or b) that has been sent. On the contrary, the ?d-labelled transition is initially disabled because the input of d is possible only if b has been sent. So, the peer interacting with a process that follows the session type S *must* necessarily consume either a or b in order to understand whether or not it can send d.

Finally, we have to motivate the *coinductive* definition of the LTS. This aspect is related to both asynchrony and fairness. Consider a session type S such that $S = \mathsf{!a}.S + \mathsf{!b}.\mathsf{?c}.T$. The question is whether or not the ?c-labelled transition should be enabled from S. In principle, S allows the output of an infinite sequence of a messages and so the ?c-labelled transition should be disabled. In this work, however, we make the assumption that this behavior is unrealistic or *unfair*, therefore we want the ?c-labelled transition to be enabled. From a technical standpoint, the only derivation that allows us to express an ?c-labelled transition for S is the following one

$$\dfrac{\dfrac{\dfrac{\vdots}{S \xrightarrow{\mathsf{?c}} S'}\,[\text{L-ASYNC}] \qquad \dfrac{}{\mathsf{?c}.T \xrightarrow{\mathsf{?c}} T}\,[\text{L-SYNC}]}{S \xrightarrow{\mathsf{?c}} S'}\,[\text{L-ASYNC}]}$$

where $S' = \mathsf{!a}.S' + \mathsf{!b}.T$. This infinite derivation is legal only if the LTS is coinductively defined.

Technically, concerning our fairness assumption, remember that in this paper we take the viewpoint that sessions are meant to eventually terminate so that neverending communication protocols are excluded. This is reflected both in the notion of session correctness (definition 3) from which refinement is derived (definition 4), as well as in the definition of the asynchronous LTS (rule [L-ASYNC]). Concerning the nature of our fairness assumption, it turns out to be an instance of what van Glabbeek and Höfner [25] call *full fairness* where the property being preserved is the reachability of the $[\mathsf{end}, \epsilon] | [\mathsf{end}, \epsilon]$ configuration.

The coinductive definition of the LTS may cast doubts on the significance of the labels of transitions, in the sense that some (input) transitions could be coinductively derivable for a given session type S even if they do not correspond to actions that are actually described

within S. For example, for the pre-session type $S = !\mathsf{a}.S$ it would be possible to derive *every* transition of the form $S \xrightarrow{?\mathsf{b}} T$ by using infinitely many applications of [L-ASYNC]. The following result rules out this possibility. The key element of the proof is the fact that, because of the well-formedness condition (3), every subtree of a session type contains a leaf of the form end.

▶ **Proposition 6.** *If $S \xrightarrow{\alpha}$, then there exists T and φ made of output actions only such that $S \xrightarrow{\varphi} T \xrightarrow{\alpha}$ and the last transition is derived by* [L-SYNC].

After looking at the transition sequences in (1), the reader may also wonder whether the LTS always satisfies the diamond property when a session type simultaneously enables both input and output actions. Crucially, this is the case.

▶ **Proposition 7.** *If $S \xrightarrow{?\mathsf{a}} S'$ and $S \xrightarrow{!\mathsf{b}} S''$, then there exists T such that $S' \xrightarrow{!\mathsf{b}} T$ and $S'' \xrightarrow{?\mathsf{a}} T$.*

Having established these basic facts about the LTS of session types, we introduce some more notation. If $\varphi = \alpha_1 \cdots \alpha_n$, we write $\varphi.S$ for $\alpha_1 \dots \alpha_n.S$. Note that $\varepsilon.S = S$. Then, we define a notion of *partial derivative* for session types that resembles Brzozowski's derivative for regular expressions [8]. In particular, if $S \xrightarrow{\varphi} T$ we say that T is the derivative of S after φ and we write $S(\varphi)$ for T. Note that $S(\varphi)$ is uniquely defined because of the well-formedness condition (1). We define the *inputs* and *outputs* of a session type as $\mathsf{inp}(S) \stackrel{\text{def}}{=} \{\mathsf{a} \mid S \xrightarrow{?\mathsf{a}}\}$ and $\mathsf{out}(S) \stackrel{\text{def}}{=} \{\mathsf{a} \mid S \xrightarrow{!\mathsf{a}}\}$. Note that $\mathsf{inp}(\mathsf{end}) = \mathsf{out}(\mathsf{end}) = \emptyset$. Occasionally we use $\mathsf{out}(\cdot)$ also as a predicate so that $\mathsf{out}(S)$ holds if and only if $\mathsf{out}(S) \neq \emptyset$. Finally, the set of *traces* of a session type is defined as $\mathsf{tr}(S) \stackrel{\text{def}}{=} \{\varphi \mid S \xrightarrow{\varphi} \mathsf{end}\}$.

▶ **Example 8.** Let us formalize the protocols of the stream and batch processing servers we have sketched in Section 1. Consider the session types

$$S = ?\mathsf{req}.!\mathsf{resp}.S + ?\mathsf{stop}.!\mathsf{stop}.\mathsf{end} \qquad \text{and} \qquad \begin{array}{l} T = ?\mathsf{req}.T + ?\mathsf{stop}.T' \\ T' = !\mathsf{resp}.T' + !\mathsf{stop}.\mathsf{end} \end{array}$$

and observe that $\mathsf{tr}(T) = \{(?\mathsf{req})^m ?\mathsf{stop} (!\mathsf{resp})^n !\mathsf{stop} \mid m, n \in \mathbb{N}\}$. Note also that $S \xrightarrow{(?\mathsf{req})^n ?\mathsf{stop}} !\mathsf{resp}^n.!\mathsf{stop}.\mathsf{end}$ for every $n \in \mathbb{N}$. Therefore, S allows all the sequences of input actions allowed by T, after which it performs a subset of the sequences of output actions allowed by T. At the same time, S allows the anticipation of output actions allowed by T. ⌟

We now define a notion of correctness that adapts, to the new semantics, the notion of correct composition used in Definition 3. A *session composition* is a pair $S \parallel T$ of session types. Session compositions reduce according to the following rule:

$$\frac{S \xrightarrow{\overline{\alpha}} S' \qquad T \xrightarrow{\alpha} T'}{S \parallel T \to S' \parallel T'} \quad \begin{array}{l} \mathsf{out}(S) \subseteq \mathsf{inp}(T) \\ \mathsf{out}(T) \subseteq \mathsf{inp}(S) \end{array} \tag{2}$$

Notice that we overload $\to$ and $\to^*$, which were previously used to denote the transitions over session configurations in Definition 1. Their actual meaning is made clear by the context. Moreover, note that a session composition is *stuck* (*i.e.* it does not reduce) if one of the two session types in the composition enables an output for which the corresponding input is disabled in the other session type.

▶ **Example 9.** Consider the types $S = !\mathsf{a}.?\mathsf{a}.\mathsf{end} + !\mathsf{b}.?\mathsf{b}.\mathsf{end}$ and $T = !\mathsf{a}.?\mathsf{a}.\mathsf{end}$ already discussed in Example 2. We have observed that their configuration $[S, \epsilon]|[T, \epsilon]$ can perform transitions according to the relation in Definition 1. On the contrary, the session composition $S \parallel T$ is stuck because $\mathsf{b} \in \mathsf{out}(S)$ while $\mathsf{b} \notin \mathsf{inp}(T)$, hence $\mathsf{out}(S) \not\subseteq \mathsf{inp}(T)$. ⌟

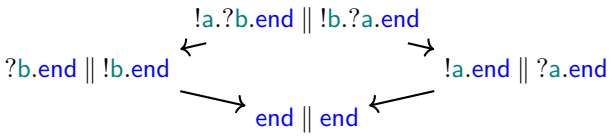
The definition of correct session composition is the same as Definition 3 adapted to the new asynchronous semantics of types. Intuitively, in a correct session composition $S \parallel T$, the only state that is allowed to be stuck is the final one, where both session types have reduced to end meaning that the session has successfully terminated. Moreover, every intermediate state $S' \parallel T'$ that is reachable from $S \parallel T$ must itself be on a path that leads to successful termination of the session.

▶ **Definition 10.** *We say that $S \parallel T$ is* correct*, notation $S \bowtie T$, if $S \parallel T \to^* S' \parallel T'$ implies that $S' \parallel T' \to^* \mathsf{end} \parallel \mathsf{end}$.*

A simple example of correct session composition is !a.?b.end ∥ ?a.!b.end, which admits only one reduction sequence

$$!a.?b.\mathsf{end} \parallel ?a.!b.\mathsf{end} \to ?b.\mathsf{end} \parallel !b.\mathsf{end} \to \mathsf{end} \parallel \mathsf{end}$$

where every intermediate state can reduce further towards successful termination. Note that a ?b-labelled transition is immediately enabled by the session type on the left hand side of the initial composition, but the matching !b-labelled transition becomes enabled only after the first reduction. More interestingly, also the composition !a.?b.end ∥ !b.?a.end is correct. In this case the composition may evolve non-deterministically in two ways, depending on which output message is consumed first:



In synchronous theories of (binary) session types duality implies correctness, to the point that in most cases session correctness is expressed in terms of duality. This does not hold for the notion of correctness given by Bravetti, Lange and Zavattaro [6] because they do not consider the well-formedness condition (3). In fact, every type without end cannot be correctly composed with any other type, including its dual. We now show that by restricting to types satisfying also the well-formedness condition (3) we recover this property which also plays a key role in the alternative characterization of subtyping discussed in the next section.

▶ **Proposition 11.** *$\overline{S} \bowtie S$ holds for every session type S.*

As discussed in Example 9, the presence of FIFO message queues in Definition 1 allows types starting with outputs to emit messages and store them in the queues without performing any check about the possibility for the receiver to consume these messages. On the contrary, the new reduction semantics of session composition (2) checks that all outputs can be consumed by the partner. Despite this difference, the two semantics can be proved "equivalent". More specifically, we will prove the following correspondence result: two session types S and T are *compliant* (according to Definition 3) if and only if $S \bowtie T$ (according to Definition 10).

In order to state the correspondence result, it is convenient to introduce some additional notation allowing us to consider the configuration queues as sequences of input actions to be executed in order to consume the buffered messages. More precisely, given the queue $\omega = \mathsf{a}_1\mathsf{a}_2 \ldots \mathsf{a}_n$, we write $?\omega$ for the corresponding sequence $?\mathsf{a}_1?\mathsf{a}_2 \ldots ?\mathsf{a}_n$ of input actions. We also use the notation $S \xrightarrow{?\omega} S'$ as a shortcut for $S \xrightarrow{?\mathsf{a}_1?\mathsf{a}_2 \ldots ?\mathsf{a}_n} S'$. Notice that the queue ω could be empty, i.e. $\omega = \epsilon$, in which case $S \xrightarrow{?\omega} S$.

In Example 2 we have seen a computation defined according to Definition 1 that cannot be mimicked by our reductions defined by rule (2). We now prove that the vice versa actually holds. More precisely we identify a way to relate session compositions to session configurations, and we prove that each reduction of a session compositions can be mimicked by a corresponding sequence of transitions of configurations. A session composition $S_1 \parallel T_1$ corresponds to several configurations $[S, \omega_S]|[T, \omega_T]$ such that $S \xrightarrow{?\omega_S} S_1$ and $T \xrightarrow{?\omega_T} T_1$. That is, S_1 can be thought of as the residual of S after all the messages in the queue ω_S of incoming messages have been consumed. Similarly for T_1 and T.

▶ **Example 12.** Consider $S = !\mathsf{a}.?\mathsf{b}.\mathsf{end}$ and $T = ?\mathsf{a}.!\mathsf{b}.\mathsf{end}$. We have $S \parallel T \to ?\mathsf{b}.\mathsf{end} \parallel !\mathsf{b}.\mathsf{end}$. Similarly, we have the following configuration transition $[S, \epsilon]|[T, \epsilon] \to [?\mathsf{b}.\mathsf{end}, \epsilon]|[T, \mathsf{a}]$. Configuration $[?\mathsf{b}.\mathsf{end}, \epsilon]|[T, \mathsf{a}]$ is one of those corresponding to $?\mathsf{b}.\mathsf{end} \parallel !\mathsf{b}.\mathsf{end}$ because of the buffered message a and the type T such that $T \xrightarrow{?\mathsf{a}} !\mathsf{b}.\mathsf{end}$. Now we have the reduction $?\mathsf{b}.\mathsf{end} \parallel !\mathsf{b}.\mathsf{end} \to \mathsf{end} \parallel \mathsf{end}$. Configuration $[?\mathsf{b}.\mathsf{end}, \epsilon]|[T, \mathsf{a}]$ can mimick this reduction with the two following transitions $[?\mathsf{b}.\mathsf{end}, \epsilon]|[T, \mathsf{a}] \to [?\mathsf{b}.\mathsf{end}, \epsilon]|[!\mathsf{b}.\mathsf{end}, \epsilon] \to [?\mathsf{b}.\mathsf{end}, \mathsf{b}]|[\mathsf{end}, \epsilon]$. The reached configuration corresponds to $\mathsf{end} \parallel \mathsf{end}$ because of the buffered message b and the type $?\mathsf{b}.\mathsf{end}$ such that $?\mathsf{b}.\mathsf{end} \xrightarrow{?\mathsf{b}} \mathsf{end}$. ⌟

▶ **Lemma 13.** *Consider the configuration $[S, \omega_S]|[T, \omega_T]$ with $S \xrightarrow{?\omega_S} S_1$ and $T \xrightarrow{?\omega_T} T_1$. If $S_1 \parallel T_1 \to S_1' \parallel T_1'$ then $[S, \omega_S]|[T, \omega_T] \to^* [S', \omega_{S'}]|[T', \omega_{T'}]$ with $S' \xrightarrow{?\omega_{S'}} S_1'$ and $T' \xrightarrow{?\omega_{T'}} T_1'$.*

We now investigate the possibility for the reduction relation defined by rule (2) to mimick sequences of transitions of corresponding configurations. This is not true in general, as shown in Example 9. However, we can prove this result under the assumption that the initial session compositions, or the initial configurations, are correct. These two cases are considered in the next two lemmas, the first of which states that a correct session composition $S \parallel T$ can mimick all computations of the configuration $[S, \epsilon]|[T, \epsilon]$ and the reached compositions/configurations are related by the correspondence relation discussed above and used in Lemma 13.

▶ **Lemma 14.** *Let $S \parallel T$ be a correct session composition. If $[S, \epsilon]|[T, \epsilon] \to^* [S', \omega_{S'}]|[T', \omega_{T'}]$ then $S \parallel T \to^* S_1 \parallel T_1$ with $S' \xrightarrow{?\omega_{S'}} S_1$ and $T' \xrightarrow{?\omega_{T'}} T_1$.*

A similar correspondence result holds also for initial correct configurations.

▶ **Lemma 15.** *Let $[S, \epsilon]|[T, \epsilon]$ be a correct configuration. Consider the transition sequence $[S, \epsilon]|[T, \epsilon] \to^* [S', \omega_{S'}]|[T', \omega_{T'}]$ and two types S_1 and T_1 s.t. $S' \xrightarrow{?\omega_{S'}} S_1$ and $T' \xrightarrow{?\omega_{T'}} T_1$. If $[S', \omega_{S'}]|[T', \omega_{T'}] \to [S'', \omega_{S''}]|[T'', \omega_{T''}]$ then $S_1 \parallel T_1 \to^* S_1' \parallel T_1'$ with $S'' \xrightarrow{?\omega_{S''}} S_1'$ and $T'' \xrightarrow{?\omega_{T''}} T_1'$.*

We can now conclude with the main result of this section, that is the correspondence between compliance (Definition 3) and correctness (Definition 10).

▶ **Theorem 16.** *We have that S and T are compliant (Definition 3) iff $S \bowtie T$ (Definition 10).*

4 Fair Asynchronous Session Subtyping

In this section we present the main contribution of the paper, namely a sound and complete characterization of the refinement relation defined by Bravetti, Lange and Zavattaro [6] recalled in Definition 4. In light of Theorem 16, such refinement can be alternatively defined using the new labelled transition system for asynchronous session types defined in Section 3 and the notion of correctness in Definition 10:

$S \preceq T$ if and only if $R \bowtie T$ implies $R \bowtie S$ for every R

This definition corresponds to the definition of a subtyping relation for session following *Liskov's substitution principle* [19], where the property to be preserved is session correctness.

We start the characterization of $\preceq$ by introducing the following coiductive relation which, as we will see later, turns out to be an overapproximation of $\preceq$.

▶ **Definition 17.** *We say that* $\mathcal{S}$ *is an* asynchronous subtyping relation *if* $(S,T) \in \mathcal{S}$ *implies:*

1. *if* $T = \mathsf{end}$, *then* $S = \mathsf{end}$;
2. *if* $T \xrightarrow{?a} T'$, *then* $S \xrightarrow{?a} S'$ *and* $(S',T') \in \mathcal{S}$;
3. *if* $\mathsf{out}(T)$, *then* $\mathsf{out}(S)$ *and* $S \xrightarrow{!a} S'$ *implies* $T \xrightarrow{!a} T'$ *and* $(S',T') \in \mathcal{S}$.

The clauses of Definition 17 specify some expected requirements for a session subtyping relation: a terminated session type end is in relation with itself only; every input action in a supertype T should be matched by the same input action allowed in the subtype S and the corresponding continuations should still be related by subtyping; dually, every output action allowed by a subtype S should be matched by an output action allowed by the supertype T and the corresponding continuations should still be related by subtyping.

Interestingly, these clauses are essentially those found in analogous characterizations of *synchronous subtyping for session types* [16], modulo the different orientation of $\preceq$ due to our viewpoint based on the substitution of processes rather than on the substitution of channels.[1] However, there are a couple of quirks that separate Definition 17 from sibling definitions. First of all, recall that transitions like $T \xrightarrow{?a} T'$ and $S \xrightarrow{?a} S'$ may concern "deep" input actions enabled by T and S, even when T and S begin with output actions. Hence, clause (2) may allow pairs of session types to be related even if they do not start with the same kind of actions. A simple instance of this fact is given by the session types $!a.?b.S$ and $?b.!a.S$ discussed earlier, for which we have $!a.?b.S \xrightarrow{?b} !a.S$ and $?b.!a.S \xrightarrow{?b} !a.S$. Another novelty is that the clauses (2) and (3) are no longer mutually exclusive, since it may happen that $\mathsf{out}(T)$ holds for a T that also allows (deep) input transitions. For example, any asynchronous subtyping relation that includes the pair $(!a.?b.S, !a.?b.S)$ must also include the pair $(?b.S, ?b.S)$ because of clause (3) as well as the pair $(!a.S, !a.S)$ because of clause (2).

▶ **Example 18.** As a first example of a non trivial asynchronous subtyping relation we consider the session types S and T presented in Example 5 which formalizes the two possible communication protocols for the ground station discussed in Section 1. We report here the definitions of the types for reader's convenience: $S = !\mathsf{tc}.S + !\mathsf{done}.S'$ where $S' = ?\mathsf{tm}.S' + ?\mathsf{over}.\mathsf{end}$ and $T = ?\mathsf{tm}.T + ?\mathsf{over}.T'$ where $T' = !\mathsf{tc}.T' + !\mathsf{done}.\mathsf{end}$. Both S and T allow sending an arbitrary number of commmands followed by a single done. Both S and T allow receiving an arbitrary number of telemetries followed by a single over. The difference is that in T the commands can only be sent after all the telemetries have been received, whereas in S the commands can be sent at any time, even before the first telemetry has been received. To see that S and T can be related by an asynchronous subtyping, observe that we have

$$\dfrac{\dfrac{\vdots}{S \xrightarrow{?\mathsf{tm}} S} \quad \dfrac{}{S' \xrightarrow{?\mathsf{tm}} S'}\,[\text{L-SYNC}]}{S \xrightarrow{?\mathsf{tm}} S}\,[\text{L-ASYNC}] \qquad \text{and} \qquad \dfrac{\dfrac{\vdots}{S \xrightarrow{?\mathsf{over}} T'} \quad \dfrac{}{S' \xrightarrow{?\mathsf{over}} \mathsf{end}}\,[\text{L-SYNC}]}{S \xrightarrow{?\mathsf{over}} T'}\,[\text{L-ASYNC}]$$

[1] The interested reader may refer to Gay [15] for a comparison of the two viewpoints.

therefore $\mathcal{S} \stackrel{\text{def}}{=} \{(S,T),(T',T'),(\mathsf{end},\mathsf{end})\}$ is an asynchronous subtyping relation. At the same time, no asynchronous subtyping relation includes the pair (T,S) since it violates the clause (3): $\mathsf{out}(S)$ holds whereas $\mathsf{out}(T)$ does not. Indeed, the session type $\overline{S}$ relies on those early outputs performed by S in order to make progress and $\overline{S} \parallel T$ is stuck. ⌟

▶ **Example 19.** Consider again the types $S = ?\mathsf{req}.!\mathsf{resp}.S + ?\mathsf{stop}.!\mathsf{stop}.\mathsf{end}$ and $T = ?\mathsf{req}.T + ?\mathsf{stop}.T'$ where $T' = !\mathsf{resp}.T' + !\mathsf{stop}.\mathsf{end}$ respectively modeling the protocols of the stream and batch processing servers defined in Example 8. To show that S and T are related by an asynchronous subtyping relation it is sufficient to show that the relation

$$\mathcal{S} \stackrel{\text{def}}{=} \{(!\mathsf{resp}^n.S, T), (!\mathsf{resp}^n.!\mathsf{stop}.\mathsf{end}, T') \mid n \in \mathbb{N}\} \cup \{(\mathsf{end},\mathsf{end})\}$$

is an asycnhronous subtyping because $(S,T) \in \mathcal{S}$. Pairs $(!\mathsf{resp}^n.S, T)$ are necessary to deal with the transition $T \xrightarrow{?\mathsf{req}} T$ which is matched by $!\mathsf{resp}^n.S \xrightarrow{?\mathsf{req}} !\mathsf{resp}^{n+1}.S$. The pairs $(!\mathsf{resp}^n.!\mathsf{stop}.\mathsf{end}, T')$ account for the transition $T \xrightarrow{?\mathsf{stop}} T'$ which is matched by $!\mathsf{resp}^n.S \xrightarrow{?\mathsf{req}} !\mathsf{resp}^n.!\mathsf{stop}.\mathsf{end}$. Notice that in pairs of the form $(!\mathsf{resp}^{n+1}.!\mathsf{stop}.\mathsf{end}, T')$ the first type has a transition $!\mathsf{resp}^{n+1}.!\mathsf{stop}.\mathsf{end} \xrightarrow{!\mathsf{resp}} !\mathsf{resp}^n.!\mathsf{stop}.\mathsf{end}$ matched by $T' \xrightarrow{!\mathsf{resp}} T'$. Finally, $!\mathsf{stop}.\mathsf{end} \xrightarrow{!\mathsf{stop}} \mathsf{end}$ is matched by $T' \xrightarrow{!\mathsf{stop}} \mathsf{end}$. ⌟

We now present a first result relating $\preceq$ and asynchronous subtyping. We have that $\preceq$ satisfies the clauses of Definition 17 hence it is an asynchronous sbtyping relation.

▶ **Theorem 20.** *$\preceq$ is an asynchronous subtyping relation.*

As we have anticipated, the largest asynchronous subtyping relation contains pairs of types that are not in subtyping relation, as illustrated by the next example.

▶ **Example 21.** Consider two variants of the batch processing server whose responses may be positive (yes) or negative (no). Their behavior is described by the session types $S = ?\mathsf{req}.!\mathsf{no}.S + ?\mathsf{stop}.!\mathsf{stop}.\mathsf{end}$ and $T = ?\mathsf{req}.(!\mathsf{yes}.T + !\mathsf{no}.T) + ?\mathsf{stop}.!\mathsf{stop}.\mathsf{end}$. The server behaving as S always responds no. The server behaving as T may respond in either way. Let $T^0 \stackrel{\text{def}}{=} T$ and $T^{n+1} \stackrel{\text{def}}{=} !\mathsf{yes}.T^n + !\mathsf{no}.T^n$. It is relatively easy to see that $\mathcal{S} \stackrel{\text{def}}{=} \{(!\mathsf{no}^n.S, T^n) \mid n \in \mathbb{N}\} \cup \{(!\mathsf{stop}.\mathsf{end}, !\mathsf{stop}.\mathsf{end}), (\mathsf{end},\mathsf{end})\}$ is an asynchronous subtyping relation, and yet $S \not\preceq T$. Indeed, consider the session type $R \stackrel{\text{def}}{=} !\mathsf{req}.R'$ where $R' = ?\mathsf{yes}.!\mathsf{stop}.\mathsf{end} + ?\mathsf{no}.R$. Then $R \bowtie T$ holds but $R \bowtie S$ does not. Basically, (a process behaving as) R insists on sending requests until it receives a positive response. At that point, it is satisfied and quits the interaction. However, only (a process behaving as) T is willing to send a positive response, whereas (a process behaving as) S is not. ⌟

In general, a purely coinductive characterization based on the clauses of Definition 17 allows us to capture the *safety-preserving properties* of subtyping, those concerning the admissibility of interactions, because they are supported by an invariant argument. However, the very same clauses do not say anything about the *liveness-preserving properties* of subtyping, those concerning the reachability of successful termination, which must be supported by a well-foundedness argument. In order to find a sound characterization of subtyping, we have to resort to *bounded coinduction* [1, 14], a particular case of coinductive definition where we consider the largest relation that satisfies the clauses in Definition 17 and that is also *included in an inductively defined relation* that preserves the reachability of successful termination. We dub this inductive relation *convergence* and we denote it by $\sqsubseteq$.

Before looking at the formal definition of $\sqsubseteq$, let us try to acquire a rough understanding of convergence by recalling Example 21, in which we have identified two session types S and T that satisfy the clauses of Definition 17 but are not related by subtyping. In that

example, we can see that the traces that lead S to termination are a subset of the traces that lead T to termination. This is a general property that holds every time S describes a more deterministic (or "less demanding") behavior compared to T. However, in the specific case of Example 21, the traces that lead S to termination are *too few*, to the point that there exists a potential partner (described by R in the example) that terminates solely relying on those traces of T that have disappeared in S. Contrast S and T those with $S' = !\mathsf{a}.!\mathsf{a}.S' + !\mathsf{b}.\mathsf{end}$ and $T' = !\mathsf{a}.T' + !\mathsf{b}.\mathsf{end}$. Also in this case S' is more deterministic than T' and some traces that lead T' to termination have disappeared in S'. Specifically, every trace of the form $\varphi = !\mathsf{a}^{2n+1}!\mathsf{b}$ is allowed by T' but not by S'. However, it is also the case that for each of these traces we can find another trace $!\mathsf{a}^{2n}!\mathsf{b}$ that shares a common prefix with φ and that leads both S' and T' to termination *after an output action* $!\mathsf{b}$. Since the behavior described by T' is always able to autonomously veer the interaction towards termination after *any number* of a messages, any session type R that can be correctly combined with T' must be prepared to receive this b message at any time, meaning that termination is preserved also if R is combined with S'.

Let us now define $\sqsubseteq$ formally.

▶ **Definition 22.** Convergence *is the relation* $\sqsubseteq$ *inductively defined by the rule:*

$$\frac{\forall \varphi \in \mathsf{tr}(T) \setminus \mathsf{tr}(S) : \exists \psi \leq \varphi, \mathsf{a} : S(\psi!\mathsf{a}) \sqsubseteq T(\psi!\mathsf{a})}{S \sqsubseteq T}$$

Intuitively, a derivation for the relation $S \sqsubseteq T$ measures the "difference" between S and T in terms of allowed traces. When $\mathsf{tr}(T) \subseteq \mathsf{tr}(S)$ there is essentially no difference between S and T, so the rule that defines $\sqsubseteq$ has no premises and turns into an axiom. When $\mathsf{tr}(T) \not\subseteq \mathsf{tr}(S)$, then for every trace φ of actions allowed by T but not by S there is a prefix $\psi \leq \varphi$ shared by both S and T and followed by an output action $!\mathsf{a}$ that leads to a pair of session types $S(\psi!\mathsf{a})$ and $T(\psi!\mathsf{a})$ that are "slightly less different" from each other in terms of allowed traces. Indeed, the derivation for $S(\psi!\mathsf{a}) \sqsubseteq T(\psi!\mathsf{a})$ must be strictly smaller than that for $S \sqsubseteq T$, or else $S \sqsubseteq T$ would not be inductively derivable.

Note that $\sqsubseteq$ is trivially reflexive, but apart from that it is generally difficult to understand when two session types are related by convergence because of the *non-local* flavor of the relation. However, it is easy to see that any two session types related by an asynchronous subtyping where at least one of them is finite are also related by convergence.

▶ **Proposition 23.** *If* $\mathcal{S}$ *is an asynchronous subtyping relation such that* $(S, T) \in \mathcal{S}$ *and at least one among* S *and* T *is finite, then* $S \sqsubseteq T$*.*

Proof. A simple induction on either S or T, depending on which one is finite. ◀

When both S and T are infinite, understanding whether $S \sqsubseteq T$ holds or not may require some non-trivial reasoning on their traces. Let us work out a few examples.

▶ **Example 24.** Let $S = !\mathsf{a}.!\mathsf{a}.S + !\mathsf{b}.\mathsf{end}$ and $T = !\mathsf{a}.T + !\mathsf{b}.\mathsf{end}$. To prove that $S \sqsubseteq T$, consider $\varphi \in \mathsf{tr}(T) \setminus \mathsf{tr}(S)$. It must be the case that $\varphi = !\mathsf{a}^{2n+1}!\mathsf{b}$ for some n. Take $\psi \stackrel{\text{def}}{=} !\mathsf{a}^{2n}$. Now $S(\psi!\mathsf{b}) = T(\psi!\mathsf{b}) = \mathsf{end}$, hence $S(\psi!\mathsf{b}) \sqsubseteq T(\psi!\mathsf{b})$ by reflexivity of $\sqsubseteq$. ⌟

▶ **Example 25.** Consider again $S = ?\mathsf{req}.!\mathsf{no}.S + ?\mathsf{stop}.!\mathsf{stop}.\mathsf{end}$ and $T = ?\mathsf{req}.(!\mathsf{yes}.T + !\mathsf{no}.T) + ?\mathsf{stop}.!\mathsf{stop}.\mathsf{end}$ from Example 21. Since we conjecture $S \not\sqsubseteq T$ we can focus on a particular $\varphi \in \mathsf{tr}(T) \setminus \mathsf{tr}(S)$, namely $\varphi = ?\mathsf{req}!\mathsf{yes}\varphi' \in \mathsf{tr}(T) \setminus \mathsf{tr}(S)$. Note that $\mathsf{out}(S)$ does not hold, so the only prefix of φ that we may reasonably consider is $\psi \stackrel{\text{def}}{=} ?\mathsf{req}$ and the only output action that S may perform after such prefix is $!\mathsf{no}$. But then $S(\psi!\mathsf{no}) = S$ and $T(\psi!\mathsf{no}) = T$, hence it is not possible to build a well-founded derivation for $S \sqsubseteq T$. We conclude $S \not\sqsubseteq T$. ⌟

▶ **Example 26.** Let us prove that the session types S and T in Example 8 satisfy the relation $S \sqsubseteq T$. Consider $\varphi \in \mathsf{tr}(T) \setminus \mathsf{tr}(S)$. It must be the case that $\varphi = ?\mathsf{req}^m?\mathsf{stop}!\mathsf{resp}^n!\mathsf{stop}$ for some $m, n \in \mathbb{N}$ with $m \neq n$. If $m < n$, then we can take $\psi \stackrel{\text{def}}{=} ?\mathsf{req}^m?\mathsf{stop}!\mathsf{resp}^m \leq \varphi$ and now $S(\psi!\mathsf{stop}) = T(\psi!\mathsf{stop}) = \mathsf{end}$ and we conclude by reflexivity of $\sqsubseteq$. If $m > n$, then we can take $\psi \stackrel{\text{def}}{=} ?\mathsf{req}^m?\mathsf{stop}!\mathsf{resp}^n \leq \varphi$ and now $S(\psi!\mathsf{resp}) = !\mathsf{resp}^{m-n-1}.!\mathsf{stop}.\mathsf{end} \sqsubseteq T' = T(\psi!\mathsf{resp})$ using Proposition 23 (because the asynchnronous subtyping relation $\mathcal{S}$ of Example 19 contains all pairs $(!\mathsf{resp}^n.!\mathsf{stop}.\mathsf{end}, T')$, for every n, and the type and the types $!\mathsf{resp}^{m-n-1}.!\mathsf{stop}.\mathsf{end}$ is finite). ⌟

We are finally ready to state our main result which confirms that $\sqsubseteq$ indeed provides the inductively defined space within which we can characterize $\preceq$ as the largest asynchronous subtyping relation. Note that the characterization is both sound and complete.

▶ **Theorem 27.** *$\preceq$ is the largest asynchronous subtyping relation included in $\sqsubseteq$.*

▶ **Example 28.** We are finally able to show that S and T in Example 8 are such that $S \preceq T$. In Example 19 we have considered the following asynchronous subtyping relation:

$$\mathcal{S} \stackrel{\text{def}}{=} \{(!\mathsf{resp}^n.S, T), (!\mathsf{resp}^n.!\mathsf{stop}.\mathsf{end}, T') \mid n \in \mathbb{N}\} \cup \{(\mathsf{end}, \mathsf{end})\}$$

We now prove that all the pairs in $\mathcal{S}$ are also included in $\sqsubseteq$, confirming that $S \preceq T$ by Theorem 27. In Example 26 we have already shown that $S \sqsubseteq T$. We can use similar arguments to show that also the pairs $(!\mathsf{resp}^n.S, T)$ are such that $!\mathsf{resp}^n.S \sqsubseteq T$. Consider $\varphi \in \mathsf{tr}(T) \setminus \mathsf{tr}(!\mathsf{resp}^n.S)$. It must be the case that $\varphi = ?\mathsf{req}^m?\mathsf{stop}!\mathsf{resp}^l!\mathsf{stop}$ for some $m, l \in \mathbb{N}$ with $m + n \neq l$. If $m + n < l$, then we can take $\psi \stackrel{\text{def}}{=} ?\mathsf{req}^m?\mathsf{stop}!\mathsf{resp}^{m+n} \leq \varphi$ and now $S(\psi!\mathsf{stop}) = T(\psi!\mathsf{stop}) = \mathsf{end}$ and we conclude by reflexivity of $\sqsubseteq$. If $m + n > l$, then we can take $\psi \stackrel{\text{def}}{=} ?\mathsf{req}^m?\mathsf{stop}!\mathsf{resp}^l \leq \varphi$ and now $S(\psi!\mathsf{resp}) = !\mathsf{resp}^{m+n-l-1}.!\mathsf{stop}.\mathsf{end} \sqsubseteq T' = T(\psi!\mathsf{resp})$ using Proposition 23. The pairs $(!\mathsf{resp}^n.!\mathsf{stop}.\mathsf{end}, T')$, $(!\mathsf{stop}.\mathsf{end}, T')$, and $(\mathsf{end}, \mathsf{end})$ all contain at least one finite type, hence they are included in $\sqsubseteq$ by Proposition 23. ⌟

5 Related Work and Concluding Remarks

Gay and Hole [16] introduced the first notion of subtyping for *synchronous* session types. This subtyping supports variance on both inputs and outputs so that a subtype can have more external nondeterminism (by adding branches in input choices) and less internal nondeterminism (by removing branches in output choices). Padovani [23] studied a notion of fair subtyping for *synchronous* multi-party session types that preserves a notion of correctness similar to our Definition 3. One key difference between Padovani's fair subtyping and Gay-Hole subtyping is that the variance of outputs must be limited in such a way that an output branch can be pruned only if it is not necessary to reach successful termination. Ciccone and Padovani [23, 11, 12] have presented sound and complete charcaterizations of fair subtyping in the synchronous case. These characterizations all combine a coinductively defined relation and an inductively defined relation (called "convergence"), which respectively capture the safety-preserving and the liveness-preserving aspects of subtyping.

Subtyping relations for asynchronous session types have been defined by Mostrous and Yoshida [21], Chen *et al.* [9] and Bravetti, Lange and Zavattaro [6]. In the asynchronous case a subtype can anticipate output actions w.r.t. its supertype because anticipated outputs can be stored in the communication queues without altering the overall interaction behaviour. The first proposal for asynchronous subtyping [21] allows a subtype to execute loops of anticipated outputs, thus delaying input actions indefinitely. This could leave orphan messages in the

buffers. The subsequent work of Chen *et al.* [9] imposes restrictions on output anticipation so as to avoid orphan messages. Bravetti, Lange and Zavattaro [6] have defined the first fair (liveness-preserving) asynchronous session subtyping along the lines of Definition 4. However, they only provided a sound (but not complete) coinductive characterization.

In this paper we present the first complete characterization of the fair asynchronous subtyping relation defined by Bravetti, Lange and Zavattaro [6] using the techniques introduced by Ciccone and Padovani in the synchronous case [23, 11, 12]. In particular, we complement a coinductively defined subtyping relation with a notion of convergence. To do so, we leverage a novel "asynchronous" operational semantics of session types in which a type can perform an input transition even if such input is prefixed by outputs, under the assumption that such input is present along *every* initial sequence of outputs.

Another fair asynchronous subtyping relation that makes use of a similar operational semantics has been recently defined by Padovani and Zavattaro [24]. Unlike the relation defined by Bravetti, Lange and Zavattaro [6] and characterized in this paper, the subtyping relation of Padovani and Zavattaro [24] preserves a liveness property that is strictly weaker than successful session termination. For this reason, their operational semantics is completely symmetric with respect to the treatment of input and output actions and no inductively-defined convergence relation is necessary in order to obtain a sound and complete characterization of subtyping.

A notable difference between the paper of Bravetti, Lange and Zavattaro [6] and our own is the fact that we focus on those session types describing protocols that can always terminate, whereas Bravetti, Lange and Zavattaro make no such assumption. This choice affects the properties of the subtyping relation. In particular, Bravetti, Lange and Zavattaro [6] show that a supertype can have additional input branches provided that such branches are *uncontrollable*. A session type is uncontrollable if there are no session types that can be correctly composed with it. Our well-formedness condition (3) in the definition of types, guarantees that all session types are controllable. We argue that the characterization presented in this paper can be easily extended to the more general case where uncontrollable session types are allowed, following the approach considered by Padovani [23]. Basically, types that do not satisfy the well-formedness condition (3) can be *normalized* by pruning away the uncontrollable subtrees. The normalization yields session types that are semantically equivalent to the original ones and that satisfy the condition (3). At that point, our characterization can be used to establish whether or not they are related by subtyping.

We envision three lines of development of this work. One line is concerned with the study of algorithmic versions of our notions of correct composition, subtyping, and convergence. As for all the known subtypings for asychronous sessions, these notions turn out to be undecidable [7]. In the literature, sound (but not complete) algorithmic characterizations have been investigated for different variants of asynchronous session subtyping [4, 5, 2, 6]. We plan to investigate the possibility to adapt these approaches to our new formalization of fair asynchronous subtyping, possibly taking advantage of the novel operational semantics for asynchronous session types. Another line of future work concerns the application of fair asynchronous subtyping to the definition of type systems ensuring successful session termination of asynchronously communicating processes. Such type systems have been studied for *synchronous* communication by Ciccone, Dagnino and Padovani [11, 10]. A first proposal of such type system for *asynchronous* communication has been recently given by Padovani and Zavattaro [24]. However, as we have pointed out above, the subtyping relation used in their type system does not preserve successful session termination. As a consequence, their type system requires a substantial amount of additional annotations in types and in

typing judgements in order to enforce successful termination. It may be the case that relying on a subtyping relation that provides stronger guarantees, like the one characterized in the present paper, could simplify the type system and possibly enlarge the family of typeable processes.

Finally, it may be worth investigating whether the characterization of fair asynchronous subtyping studied in this paper for *binary* session types extends smoothly to *multiparty* session types as well. In the synchronous setting, it is known that there are no substantial differences between the binary and the multiparty case, except for the presence of *role names* in session types [23, 11]. We conjecture that an asynchronous LTS for multiparty session types can be defined in a very natural way along the lines presented in this paper. However, the multiparty case grants additional out-of-order transitions compared to the binary case (see the *unfair* multiparty asynchronous subtyping defined by Ghilezan *et al.* [17]) and working out all of the resulting technicalities could be challenging.

References

1 Davide Ancona, Francesco Dagnino, and Elena Zucca. Generalizing inference systems by coaxioms. In Hongseok Yang, editor, *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, volume 10201 of *Lecture Notes in Computer Science*, pages 29–55. Springer, 2017. doi:10.1007/978-3-662-54434-1_2.

2 Laura Bocchi, Andy King, and Maurizio Murgia. Asynchronous subtyping by trace relaxation. In Bernd Finkbeiner and Laura Kovács, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 30th International Conference, TACAS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part I*, volume 14570 of *Lecture Notes in Computer Science*, pages 207–226. Springer, 2024. doi:10.1007/978-3-031-57246-3_12.

3 Daniel Brand and Pitro Zafiropulo. On communicating finite-state machines. *J. ACM*, 30(2):323–342, 1983. doi:10.1145/322374.322380.

4 Mario Bravetti, Marco Carbone, Julien Lange, Nobuko Yoshida, and Gianluigi Zavattaro. A sound algorithm for asynchronous session subtyping. In *CONCUR*, volume 140 of *LIPIcs*, pages 38:1–38:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPICS.CONCUR.2019.38.

5 Mario Bravetti, Marco Carbone, Julien Lange, Nobuko Yoshida, and Gianluigi Zavattaro. A sound algorithm for asynchronous session subtyping and its implementation. *Log. Methods Comput. Sci.*, 17(1), 2021. URL: https://lmcs.episciences.org/7238.

6 Mario Bravetti, Julien Lange, and Gianluigi Zavattaro. Fair asynchronous session subtyping. *Log. Methods Comput. Sci.*, 20(4), 2024. doi:10.46298/LMCS-20(4:5)2024.

7 Mario Bravetti, Luca Padovani, and Gianluigi Zavattaro. A Sound and Complete Characterization of Fair Asynchronous Session Subtyping. Technical report, University of Bologna, 2025. doi:10.48550/arXiv.2506.06078.

8 Janusz A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, 1964. doi:10.1145/321239.321249.

9 Tzu-Chun Chen, Mariangiola Dezani-Ciancaglini, Alceste Scalas, and Nobuko Yoshida. On the preciseness of subtyping in session types. *Log. Methods Comput. Sci.*, 13(2), 2017. doi:10.23638/LMCS-13(2:12)2017.

10 Luca Ciccone, Francesco Dagnino, and Luca Padovani. Fair termination of multiparty sessions. *J. Log. Algebraic Methods Program.*, 139:100964, 2024. doi:10.1016/J.JLAMP.2024.100964.

11 Luca Ciccone and Luca Padovani. Fair termination of binary sessions. *Proc. ACM Program. Lang.*, 6(POPL):1–30, 2022. doi:10.1145/3498666.

12 Luca Ciccone and Luca Padovani. Inference systems with corules for combined safety and liveness properties of binary session types. *Log. Methods Comput. Sci.*, 18(3), 2022. doi:10.46298/LMCS-18(3:27)2022.

13 Bruno Courcelle. Fundamental properties of infinite trees. *Theor. Comput. Sci.*, 25:95–169, 1983. doi:10.1016/0304-3975(83)90059-2.

14 Francesco Dagnino. Coaxioms: flexible coinductive definitions by inference systems. *Log. Methods Comput. Sci.*, 15(1), 2019. doi:10.23638/LMCS-15(1:26)2019.

15 Simon J. Gay. Subtyping supports safe session substitution. In Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella, editors, *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, volume 9600 of *Lecture Notes in Computer Science*, pages 95–108. Springer, 2016. doi:10.1007/978-3-319-30936-1_5.

16 Simon J. Gay and Malcolm Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2-3):191–225, 2005. doi:10.1007/S00236-005-0177-Z.

17 Silvia Ghilezan, Jovanka Pantovic, Ivan Prokic, Alceste Scalas, and Nobuko Yoshida. Precise subtyping for asynchronous multiparty sessions. *Proc. ACM Program. Lang.*, 5(POPL):1–28, 2021. doi:10.1145/3434297.

18 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *J. ACM*, 63(1):9, 2016. doi:10.1145/2827695.

19 Barbara Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, 1994. doi:10.1145/197320.197383.

20 Robin Milner. *Communication and concurrency*. PHI Series in computer science. Prentice Hall, 1989.

21 Dimitris Mostrous and Nobuko Yoshida. Session typing and asynchronous subtyping for the higher-order π-calculus. *Inf. Comput.*, 241:227–263, 2015. doi:10.1016/J.IC.2015.02.002.

22 Dimitris Mostrous, Nobuko Yoshida, and Kohei Honda. Global principal typing in partially commutative asynchronous sessions. In *ESOP*, volume 5502 of *Lecture Notes in Computer Science*, pages 316–332. Springer, 2009. doi:10.1007/978-3-642-00590-9_23.

23 Luca Padovani. Fair subtyping for multi-party session types. *Math. Struct. Comput. Sci.*, 26(3):424–464, 2016. doi:10.1017/S096012951400022X.

24 Luca Padovani and Gianluigi Zavattaro. Fair termination of asynchronous binary sessions, 2025. doi:10.48550/arXiv.2503.07273.

25 Rob van Glabbeek and Peter Höfner. Progress, justness, and fairness. *ACM Comput. Surv.*, 52(4):69:1–69:38, 2019. doi:10.1145/3329125.

The Non-Cooperative Rational Synthesis Problem for SPEs and ω-Regular Objectives

Véronique Bruyère
Université de Mons, UMONS, Belgium

Jean-François Raskin
Université libre de Bruxelles, ULB, Belgium

Alexis Reynouard
Université libre de Bruxelles, ULB, Belgium

Marie Van Den Bogaard
Université Gustave Eiffel, CNRS, LIGM, Marne-la-Vallée, France

Abstract

This paper studies the rational synthesis problem for multi-player games played on graphs when rational players are following subgame perfect equilibria. In these games, one player, the system, declares his strategy upfront, and the other players, composing the environment, then rationally respond by playing strategies forming a subgame perfect equilibrium. We study the complexity of the rational synthesis problem when the players have ω-regular objectives encoded as parity objectives. Our algorithm is based on an encoding into a three-player game with imperfect information, showing that the problem is in 2ExpTime. When the number of environment players is fixed, the problem is in ExpTime and is NP- and coNP-hard. Moreover, for a fixed number of players and reachability objectives, we get a polynomial algorithm.

2012 ACM Subject Classification Theory of computation

Keywords and phrases non-zero-sum games, subgame perfect equilibria, rational synthesis

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2025.12

Related Version *Full Version*: `The Non-Cooperative Rational Synthesis Problem for Subgame Perfect Equilibria and ω-Regular Objectives` [19]

Funding *Véronique Bruyère*: supported by FNRS under PDR Grant T.0023.22 (Rational).
Jean-François Raskin: supported by Fondation ULB (`https://www.fondationulb.be/en/`), the Thelam fund 2024-F2150080-0021312 (Open Problems on the Decidability and Computational Complexity of Infinite Duration Games), and by FNRS under PDR Grant T.0023.22 (Rational).

1 Introduction

Studying non zero-sum games played on graphs of infinite duration with multiple players [4, 13] poses both theoretical and algorithmic challenges. This paper primarily addresses the (non-cooperative) rational synthesis problem for n-player non zero-sum games featuring *ω-regular* objectives. In this context, the goal is to algorithmically determine the existence of a strategy σ_0 for the system (also called player 0) to enforce his objective against any *rational* response from the environment (players $1, \ldots, n$). So, *rational synthesis* supports the automatic synthesis of systems wherein the environment consists of multiple agents having each their *own* objective. These agents are assumed to act rationally towards their own objective rather than being *strictly antagonistic* (to the system). This approach contrasts with the simpler scenario of zero-sum two-player game graphs, the fully antagonistic setting, a framework extensively explored in earlier reactive synthesis research, see [2] and the numerous references therein. While the computational complexity of rational synthesis, where *rationality* is defined by the concept of *Nash equilibrium* (NE), has been explored in [26], this paper revisits

© Véronique Bruyère, Jean-François Raskin, Alexis Reynouard, and Marie Van Den Bogaard;
licensed under Creative Commons License CC-BY 4.0

the rational synthesis problem using the more encompassing definition of *subgame perfect equilibrium* (SPE) to formalize rationality. Nash equilibria (NEs) have a known limitation in *sequential* games, including the infinite-duration games on graphs that we consider here: they are prone to *non-credible threats*. Such threats involve decisions within subgames (potentially reached after a deviation from the equilibrium) that, while not rational, are intended to coerce other players into specific behaviors. To address this limitation, the concept of subgame-perfect equilibrium (SPE) was introduced, as discussed in [39]. An SPE is a profile of strategies that form Nash equilibria in every subgame, thereby preventing non-rational and thus non-credible threats. Although SPEs align more intuitively with sequential games, their algorithmic treatment in the context of infinite-duration graph games remains underdeveloped. This gap persists primarily because SPEs require more complex algorithmic techniques compared to NEs. Moreover, the standard backward induction method used in *finite* duration sequential games [33] cannot be directly applied to *infinite* duration games due to the non-terminating nature of these interactions.

Kupferman et al. introduced rational synthesis in two distinct forms. The first approach, dubbed *cooperative* rational synthesis [29], considers an environment working collaboratively with the system to determine whether a specific SPE that ensures the specification of the system is met. So, in this model, the agents of the environment engage in an SPE that guarantees a win for player 0, provided such an equilibrium exists. At the opposite, the second approach, termed *non-cooperative* rational synthesis (NCRS) [34], grants the environment greater flexibility. Here, the system first selects a fixed strategy σ_0 and then the environmental agents respond by selecting any strategy that form an SPE with the fixed strategy σ_0 of the system. The central algorithmic challenge is determining whether there exists a strategy σ_0 for the system such that every resulting σ_0-fixed SPE ensures the specification of the system is upheld (SPE-NCRS problem). The computational complexity of decision problems for the cooperative synthesis problem with subgame-perfect equilibria (SPE) is now well understood. While the decidability of this problem was first established in [43], its exact complexity was resolved in [6], where the problem was shown to be NP-complete for parity objectives.

In contrast, the computational complexity of the SPE-NCRS problem remains less thoroughly investigated. Although the decidability of this problem can be determined through an encoding in Strategy Logic [38], such an encoding does not provide clear insights into the effective construction of the strategy σ_0 and is suboptimal from an algorithmic standpoint. For example, for rational environment behaviors modeled by Nash equilibria (NEs) instead of SPEs, the NE-NCRS problem can be solved in 3ExpTime for LTL objectives using a Strategy Logic encoding [34], while it can be solved in 2ExpTime through the use of tree automata [35]. This shows that using a reduction to Strategy Logic does not deliver optimal worst-case complexity. Addressing this gap, our contribution is twofold. First, we introduce an *innovative* algorithm that transforms the SPE-NCRS problem into a three-player imperfect information game. Such games were analyzed in [22], where computationally optimal algorithms for their analysis were presented. Second, our reduction offers a clear advantage over the Strategy Logic encoding, providing improved complexity (double exponential time as opposed to triple exponential time for LTL objectives in [34]). It also enables a precise analysis of the algorithmic complexity when the number of players is *fixed*, a consideration that is practically relevant in cases where the environment consists of a limited number of players.

Technical contributions and sketch of our solution. The summary of our technical contributions is depicted in Figure 1. Our main result shows how to transform the SPE-NCRS problem into a two-player zero-sum parity game with perfect information (Theorem 11), a

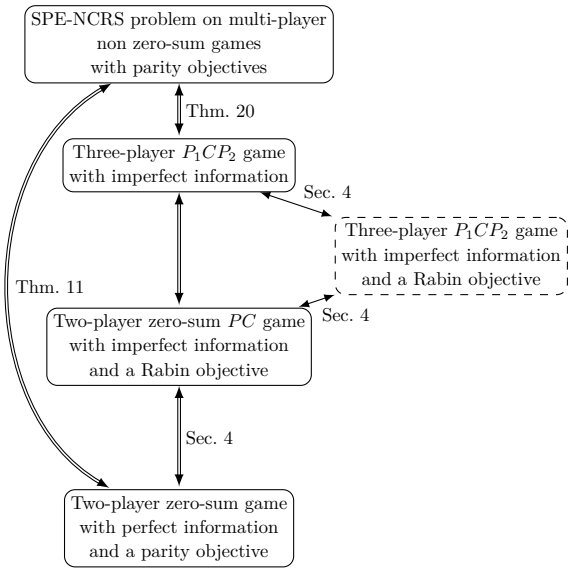


Figure 1 Structure of the article.

well-studied class of games for which algorithms are available. (Thus, our algorithm supports LTL specifications via their translation into deterministic parity automata, yielding parity conditions on the game graph). The transformation is structured in several non-trivial steps.

First, to solve the SPE-NCRS problem, we propose to use the *Prover-Challenger* framework initially introduced for the development of algorithms capable of determining the presence of a simulation relation between transition systems (for more details and extensions of this concept, see e.g. [1, 32, 37]). However, in our context, we need to use *two Provers* P_1 and P_2 with the Challenger C: P_1 aims to demonstrate the existence of a solution, i.e., a strategy σ_0 for the system, to the SPE-NCRS problem, while C seeks to counter this assertion, i.e., with a subgame perfect response $\bar{\sigma}_{-0}$ that results in an unfavorable outcome for player 0. Then, P_2 endeavors to demonstrate that the combined profile $(\sigma_0, \bar{\sigma}_{-0})$ is either not a 0-fixed SPE or its outcome favors player 0. To ensure that σ_0 is fixed and cannot be modified in subgames, we prevent P_1 from adjusting his strategy based on the interactions between C and P_2 by imposing to him *imperfect information* of the game. More intuition and the formal definition of this P_1CP_2 game are given in Section 3 together with a proof of correctness (Theorem 20).

Second, we detail our method for solving the P_1CP_2 game. Given that this game involves three players with imperfect information, specialized techniques are essential for its resolution as multi-player games with imperfect information are undecidable in general (see e.g. [40]). We employ a solution specifically adapted for our context, derived from a transformation introduced in [22]. This transformation was originally proposed for addressing similar types of three-player games with imperfect information. Due to the intricate winning condition present in our Prover-Challenger reduction, we translate it into an explicit Rabin objective. After that transformation of the winning condition, the aforementioned techniques for three-player games with imperfect information can be adapted. As a result of this transformation, we obtain a more conventional two-player zero-sum Rabin game with imperfect information.

Third, techniques to remove imperfect information, see e.g. [41, 42], can then be used to obtain the desired two-player zero-sum parity game with perfect information. Therefore, solving the SPE-NCRS problem reduces to solving this two-player zero-sum parity game for which algorithms are well-known.

In Section 4, we provide a complexity analysis of each step of our construction. This analysis enables us to derive detailed complexity results: solving the SPE-NCRS problem for parity objectives is exponential in the size of the graph and the number of priorities used by the parity objectives, and double-exponential in the number of players (while a PSpace lower bound can be deduced from [26]). Consequently, when the number of players is fixed, we achieve membership in ExpTime. Furthermore, by adapting proofs from the NE-NCRS problem [27], we establish NP- and co-NP-hardness (the NE-NCRS problem is in PSpace and both NP- and co-NP-hard, even for a fixed number of players). Finally, for the specific case of reachability objectives, we obtain a polynomial-time algorithm when the number of players is fixed, as is the case for the NE-NCRS problem [27].

To help the reader navigate the paper, we provide in Figure 1 a visual representation of its structure. Furthermore, for additional details on concepts and proofs, we provide a complete full version in [19] and two Appendices A and B to present a few helpful examples.

Related work. Recent literature, such as surveys in [4, 13, 14], underscores a growing interest in non zero-sum games for synthesis and design of multi-agents systems. Algorithms have been developed for reasoning about NEs in graph games, both for ω-regular [44] and quantitative objectives [12], or even in a concurrent setting [3]. The concept of secure equilibrium, a refinement of NE, was introduced in [25] and its potential for synthesis was demonstrated in later studies [24]. Similarly, doomsday equilibria, an expansion of secure equilibria for n-player games, is elaborated upon in [23]. All those results are related to the notion of NE. Algorithms to reason on SPEs are more recent and usually more intricate. For that reason, progresses on weaker notions like weak SPE [8, 10, 18] have been needed before establishing tight complexity bounds for SPEs. For SPEs, the exact complexity for the constrained existence for reachability games was established in [9], for parity games in [6], and for mean-payoff games in [5]. All those works introduce new algorithms that are substantially more sophisticated than those needed to study the notion of NE. As mentioned previously, the SPE-NCRS problem with LTL objectives can be addressed by reducing it to the model-checking problem of Strategy Logic, as shown in [34]. However, this encoding results in a triple-exponential complexity, even when the number of players is fixed. Consequently, this approach does not allow for a fine-grained complexity analysis when the number of players is treated as a fixed parameter, and so it cannot yield the ExpTime complexity we have achieved in this case, nor the PTime complexity for a fixed number of players with reachability objectives. Cooperative rational synthesis was first introduced in [29]. The adversarial version was later introduced in [34]. In both cases, as mentioned above, the decidability results were given through a reduction to Strategy Logic [38]. A more detailed analysis of the complexity for a large variety of ω-regular objectives and for rationality expressed as NEs was given in [26] for turn-based games and in [27] for concurrent games, for LTL objectives in [35], and for two players and for mean-payoff and discounted-sum objectives in [28]. Those results do not cover SPEs as we do in this paper. *Rational verification* (instead of rational synthesis) studies the problem of verifying that a given strategy σ_0 for the system is a solution to the NE/SPE-NCRS problem. The complexity of this simpler problem has been studied for several kinds of objectives in [7, 30, 31]. Another notion of rational environment behavior treats the environment as a single agent but with multiple, sometimes conflicting, goals, aiming for behaviors that achieve a *Pareto-optimal* balance among these objectives. Both rational synthesis and verification have been very recently studied for this concept of rationality [11, 15, 16, 17].

2 Preliminaries

In this section, we recall the necessary notions and concepts underpinning this work: First, we specify the model of games on graphs that we study, along with the corresponding definitions of plays, (winning) strategies, and objectives. Second, we present the solution concepts relevant to our non-zero-sum model and specific problem, ranging from Nash equilibria to 0-fixed subgame-perfect equilibria. Finally, we provide a precise statement of the SPE-NCRS problem (see Definition 10) and our main results (see Theorem 11).

▶ **Definition 1** (Game structure). *A* game structure *is a tuple* $G = (V, A, \Pi, \delta, v_0)$*, where:*

- $\Pi = \{0, \dots, n\}$ *is a finite set of* players*,*
- $A = \bigcup_{i=0}^{n} A_i$ *is the set of* actions *of the players, such that* A_i *is the action set of player* i*,* $i \in \Pi$*, and* $A_0 \cap (\bigcup_{i \neq 0} A_i) = \varnothing$*,*
- $V = \bigcup_{i=0}^{n} V_i$ *is the set of* states*, such that* V_i *is the state set of player* i*,* $i \in \Pi$*, and* $V_i \cap V_j = \varnothing$ *for all* $i \neq j$*,*
- $v_0 \in V$ *is the* initial state*,*
- $\delta : \bigcup_{i=0}^{n} (V_i \times A_i) \to V$ *is a partial function called the* transition function*, such that:*
  1. *G is* deadlock-free*: for every state $v \in V_i$, there exists $a \in A_i$ an action of player i, such that $\delta(v, a)$ is defined,*
  2. *G is* action-unique*: for every state $v \in V_i$, and for all $a, b \in A_i$ actions of player i, we have $\delta(v, a) = \delta(v, b) \iff a = b$.*

The size *of a game structure is given by the numbers $|V|$, $|A|$, and $|\Pi|$ of its vertices, actions, and players respectively.*

We say that a state $v \in V_i$ is *controlled* or *owned* by player i. Note that the condition $A_0 \cap (\bigcup_{i \neq 0} A_i) = \varnothing$ means that one knows from the actions when player 0 is the one that is playing. This property of the action set of player 0 is not classical nor necessary but will reveal itself useful in the remaining of the paper. Indeed, in the setting we study here, player 0 has a distinguished role compared to the other players, see Section 3. Condition 1 on the transition function ensures that in every state, there is always a possible action to play. Condition 2 requires that a transition between two states can only be achieved via a unique action.

A *play* in a game structure G is an infinite sequence $\rho \in (VA)^\omega$ of states and actions, such that ρ is of the form $v_0 a_0 v_1 a_1 \cdots$ with v_0 the initial state and where for every $k \in \mathbb{N}$, we have $\delta(v_k, a_k) = v_{k+1}$. The set of all plays in G is denoted Plays_G. A *history* is a finite prefix $h \in (VA)^* V$ of a play ending in a state of G, that we describe as $h = v_0 a_0 v_1 a_1 \dots v_k$. The set of all histories in G is denoted Hist_G, while for $i \in \Pi$, the set of histories ending in a state controlled by player i is denoted Hist_G^i. We write $h \sqsubset \rho$ if the history h is a prefix of the play ρ. We also use both notations $\sqsubset$ and $\sqsubseteq$ for two histories.

A *strategy* for player $i \in \Pi$ is a function $\sigma : \mathsf{Hist}_G^i \to A_i$, that prescribes an action $\sigma(h)$ for player i to choose for every history where it is his turn to play. The set of all strategies σ for player i is denoted by Σ_i. A collection $\bar{\sigma} = (\sigma_i)_{i \in \Pi}$ of strategies, one for each player, is called a *profile* of strategies. A play $\rho = v_0 a_0 v_1 a_1 \dots$ is *compatible* with a strategy σ_i of player i if for every $k \in \mathbb{N}$ such that $v_k \in V_i$, we have $a_k = \sigma_i(v_0 a_0 \dots v_k)$. Histories compatible with a strategy are defined similarly. A strategy σ is said to be *memoryless* when the prescribed action only depends on the last visited state, that is, for every $hav, h'a'v \in \mathsf{Hist}_G^i$, we have $\sigma(hav) = \sigma(h'a'v)$. Given a profile of strategies $\bar{\sigma} = (\sigma_i)_{i \in \Pi}$, there is a unique play starting in v_0 that is compatible with every strategy of the profile, that we call the *outcome* of $\bar{\sigma}$ and denote it by $\langle \bar{\sigma} \rangle_{v_0}$. Given a strict subset of players $\Pi' \subset \Pi$, we write $\bar{\sigma}_{-\Pi'}$ to refer to a

partial profile of strategies that contains a strategy for each player except the ones in Π'. In particular, we will often focus on the strategy of one player, say $\Pi' = \{i\}$, and the profile of strategies of the rest of the players, and use the notation $(\sigma_i, \bar{\sigma}_{-i})$ to denote the complete profile of strategies.

▶ **Definition 2** (Objective and game). *A* winning condition*, or* objective *for player i is a subset $W_i \subseteq \mathsf{Plays}_G$ of plays in the game structure G. We say that a play ρ is* winning *for player i or* satisfies *his objective if $\rho \in W_i$. Otherwise, we say that ρ is* losing *for player i. A* game *is a pair $\mathcal{G} = (G, (W_i)_{i\in\Pi})$ consisting in a game structure G, together with a profile of objectives for the players. When the context is clear, we often only write $\mathcal{G}$ to designate a game. Given a strategy profile $\bar{\sigma}$, its* gain profile *(or simply* gain*) is the Boolean vector $\bar{g}$ such that for all $i \in \Pi$, we have $g_i = 1$ if $\langle\bar{\sigma}\rangle_{v_0} \in W_i$, and $g_i = 0$ if $\langle\bar{\sigma}\rangle_{v_0} \notin W_i$. We also say that $\bar{g}$ is the gain profile of the outcome of $\bar{\sigma}$.*

In this paper, we consider the concept of *parity* objective (which is expressively complete for ω-regular objectives), as well as the particular case of *reachability* objective. Both concepts are defined in the following way:

- For each $i \in \Pi$, let $T_i \subseteq V$ be a *target set*. The reachability objective for player i is then $W_i = \{\rho = v_0 a_0 v_1 a_1 \ldots \in \mathsf{Plays}_G \mid \exists k \in \mathbb{N}, v_k \in T_i\}$ that we also denote by $\mathrm{Reach}(T_i)$. Given such a profile of target sets, we say that $(G, (\mathrm{Reach}(T_i))_{i\in\Pi})$ is a *reachability game*.
- For each $i \in \Pi$, let $C_i = \{0, 1, \ldots, d_i\}$ be a finite set of *priorities* and $\alpha_i : V \to C_i$ be a *priority function*, that is, a function that assigns a priority to each state of the game. The parity objective for player i is then $W_i = \{\rho \in \mathsf{Plays}_G \mid \min_{v\in\mathrm{Inf}(\rho)}(\alpha_i(v)) \text{ is even}\}$[1] that we also denote by $\mathrm{Parity}(\alpha_i)$. Given such a profile of priority functions, we say that $(G, (\mathrm{Parity}(\alpha_i))_{i\in\Pi})$ is a *parity game*. The *size* of each parity objective, denoted by $|\alpha_i|$, is the maximum priority d_i.

▶ Remark 3. Note that the objectives we consider in this work only put constraints on the states of the plays (and not on the actions).

▶ **Definition 4** (Winning strategy). *Let G be a game structure and W_i be an objective for player i. A strategy σ_i of player i is* winning *for W_i, if for every profile $\bar{\sigma}_{-i}$, the outcome ρ of the profile $(\sigma_i, \bar{\sigma}_{-i})$ is winning for player i, that is, $\rho \in W_i$.*

This definition focuses on W_i only. A winning strategy σ_i ensures that player i satisfies his objective W_i against any strategy profile $\bar{\sigma}_{-i}$ of the other players. In particular, it ensures that player i wins even if the other players are *strictly antagonistic*, or *adversarial*, that is, their objective is to make player i lose. This context corresponds to the classical *zero-sum* setting, and we use notation $\mathcal{G} = (G, W_i)$ for the game, since the objective W_j, for all players $j \neq i$, is implicitly defined as $\mathsf{Plays}_G \setminus W_i$. An example of reachability game and a discussion on the winning strategies of both players are given in Appendix A.

Within the classical *zero-sum* context, the *synthesis problem* asks if there exists, for a particular player, a winning strategy (see [2] for an introduction). As recalled in the preliminaries (cf. Definition 4), such a strategy ensures that his player wins against all possible strategies of the other players. However, when we depart from this adversarial hypothesis and consider the richer setting of games that are *non zero-sum*, that is, where each player has his own objective, which may overlap with the objectives of the others, the solution concept of winning strategies shows its limits. Hence the call to solution concepts such as *Nash equilibrium* or *subgame-perfect equilibrium*, on which we focus in this work.

[1] For a play $\rho = v_0 a_0 v_1 a_1 \ldots$, notation $\mathrm{Inf}(\rho)$ means $\{v \in V \mid \exists^\infty\ k,\ v_k = v\}$.

▶ **Definition 5** (Nash equilibrium [39]). *Let $\mathcal{G} = (G, (W_i)_{i\in\Pi})$ be a game and $\bar{\sigma} = (\sigma_i)_{i\in\Pi}$ be a strategy profile. We say that $\bar{\sigma}$ is a* Nash equilibrium *(NE for short) if for every player $i \in \Pi$ and strategy $\sigma'_i \in \Sigma_i$, we have $\langle \sigma'_i, \bar{\sigma}_{-i}\rangle_{v_0} \in W_i \implies \langle \sigma_i, \bar{\sigma}_{-i}\rangle_{v_0} \in W_i$.*

In other words, no player has an incentive to *deviate* unilaterally from its fixed strategy σ_i in the profile $\bar{\sigma}$: if he does so, the resulting outcome will not satisfy his objective if the outcome of $\bar{\sigma}$ was not already doing so. Notice that, however, there is no constraint on what players are allowed to do once a deviation has already occurred in the outcome of the NE. In fact, players may change their initial behaviours for completely adversarial choices. That is, NEs allow players to forget about their own objectives once the equilibrium outcome has been left, which can result in *non-credible threats*: promise of irrational behavior with respect to their own objectives. These non-credible threats are one important limitation of NEs as a solution concept that captures rationality of the players in sequential games. In order to enforce rationality in every scenario, even the ones that stem from deviations, one needs to monitor what the strategy profile prescribes after *every* history. This is exactly what *subgame perfect equilibria* (defined below) do [39]. Let $\mathcal{G} = (G, (W_i)_{i\in\Pi})$ be a game. To each history $hv \in \mathsf{Hist}_G$, with $v \in V$, corresponds a *subgame* $\mathcal{G}_{\restriction hv}$ that is the game $\mathcal{G}$ starting *after the history* hv: its plays ρ' start at the initial state v and for all $i \in \Pi$, ρ' is winning for player i if, and only if, $h\rho' \in W_i$. Given a strategy σ_i for player i, this strategy in $\mathcal{G}_{\restriction hv}$ is defined as $\sigma_{i,\restriction hv}(h') = \sigma_i(hh')$ for all histories $h' \in \mathsf{Hist}_G$ starting with the initial vertex v. Given a strategy profile $\bar{\sigma}$, we denote by $\bar{\sigma}_{\restriction hv}$ the profile $(\sigma_{i,\restriction hv})_{i\in\Pi}$ (note that its outcome starts in v).

▶ **Definition 6** (Subgame perfect equilibrium). *Let $\mathcal{G} = (G, (W_i)_{i\in\Pi})$ be a game. A* subgame perfect equilibrium *(SPE for short) is a profile $\bar{\sigma}$ of strategies such that $\bar{\sigma}_{\restriction hv}$ is an NE in each subgame $\mathcal{G}_{\restriction hv}$ of G.*

We illustrate both concepts of NE and SPE on an example in Appendix A.

In this work, we focus on *synthesizing* a strategy for a specific player (see Definition 10), referred to as the system (or player 0), which we assume to be trustworthy or whose strategy is fixed by design. For instance, this assumption is justified in scenarios where the system is implemented as a program that is subsequently used by its environment (the other players). Since the program is fixed, the system cannot deviate from its prescribed strategy. As stated at the beginning of this section, we consider the *non zero-sum* setting, where each player has their own objective, which may partially overlap with the objectives of other players. Consequently, we seek a solution concept that addresses two key aspects: (1) the specific player (the system) will not deviate from the prescribed strategy, and (2) the other players may exhibit *adversarial* behavior toward the specific player, as long as such behavior does not compromise their own objectives, given that they are assumed to be rational [34]. We next define 0-*fixed* equilibria, which meet these requirements.

▶ **Definition 7** (0-Fixed NE). *Let $\mathcal{G}$ be a game. A strategy profile $\bar{\sigma} = (\sigma_0, \bar{\sigma}_{-0})$ is a* 0-fixed NE*, if for every player i, where $i \neq 0$, and every strategy $\sigma'_i \in \Sigma_i$, we have $\langle \sigma'_i, \bar{\sigma}_{-i}\rangle_{v_0} \in W_i \implies \langle \sigma_i, \bar{\sigma}_{-i}\rangle_{v_0} \in W_i$.*

In other words, if the strategy σ_0 of player 0 is fixed, that is, if we assume player 0 *will stick* to his strategy, no other player has an incentive to deviate unilaterally. We also write that a strategy profile is a σ_0-*fixed* NE to insist on the fixed strategy σ_0 of player 0.

In this paper, we focus on 0-fixed SPEs: profiles $\bar{\sigma}$ of strategies that are 0-fixed NE in each subgame of $\mathcal{G}$ compatible with the fixed strategy of player 0. Formally:

▶ **Definition 8** (0-Fixed SPE). *Let $\mathcal{G}$ be a game. A strategy profile $\bar{\sigma} = (\sigma_0, \bar{\sigma}_{-0})$ is a* 0-fixed SPE if for every history hv compatible with σ_0, the profile $\bar{\sigma}_{\restriction hv}$ is a σ_0-fixed NE is the subgame $\mathcal{G}_{\restriction hv}$.

As for NEs, we use the terminology of σ_0-fixed SPE. Furthermore, given a strategy σ_0, we say that the strategy profile $\bar{\sigma}_{-0}$ is a *subgame-perfect response* to σ_0 if the complete profile $(\sigma_0, \bar{\sigma}_{-0})$ is a 0-fixed SPE. The next theorem guarantees the existence of an SPE in every reachability or parity game. This result also holds for 0-fixed SPEs [43].

▶ **Theorem 9** (Existence of (0-fixed) SPEs). *Given a game $\mathcal{G} = (G, (W_i)_{i\in\Pi})$ that is a parity game or a reachability game, there always exists an SPE in $\mathcal{G}$, and, for every strategy σ_0 of player* 0*, there always exists a σ_0-fixed SPE in $\mathcal{G}$.*

In this paper, we study the following synthesis problem. Intuitively, the *non-cooperative rational synthesis problem* asks, for a distinguished player, if there exists a strategy that is *as close as possible* to a winning strategy, taking into account the fact that the others are behaving rationally. The solutions seeked in this problem are indeed strategies that ensure winning for *every* possible way to complete the profile in a rational manner (i.e., other players do not cooperate but take care of their own objectives first).

▶ **Definition 10** (Non-cooperative rational synthesis problem). *The* non-cooperative rational synthesis problem *(SPE-NCRS problem for short), asks, given a game $\mathcal{G} = (G, (W_i)_{i\in\Pi})$, if there exists a strategy σ_0 of player* 0*, such that, for every subgame-perfect response $\bar{\sigma}_{-0}$, the resulting outcome $\langle(\sigma_0, \bar{\sigma}_{-0})\rangle_{v_0}$ is winning for player* 0*.*

The previous definition is illustrated with an example in Appendix A.
We now state our main results.

▶ **Theorem 11** (Complexity of the SPE-NCRS problem). *The SPE-NCRS problem for parity games is in 2ExpTime and PSpace-hard. Given $\mathcal{G} = (G, (\mathrm{Parity}(\alpha_i))_{i\in\Pi})$, this problem is solvable in time exponential in $|V|$ and each $|\alpha_i|$, $i \in \Pi$, and double-exponential in $|\Pi|$.*

Furthermore, if the number of players is fixed, the SPE-NCRS problem is in ExpTime, and NP-hard and co-NP-hard for parity games; and it is solvable in time polynomial in $|V|$ for reachability games.

3 SPE-NCRS Problem and P_1CP_2 Game

3.1 Overview of our solution

One may notice the particular role *player* 0 has in the setting of the SPE-NCRS problem: one needs to find, if it exists, a strategy σ_0 of player 0 that ensures him to win *for all* possible partial profiles $\bar{\sigma}_{-0}$ such that $(\sigma_0, \bar{\sigma}_{-0})$ is a 0-fixed SPE. Our approach is to construct, in the spirit of [36], a zero-sum two-player *Prover-Challenger* game that will, once solved, deliver the answer. The key idea in this approach is that Prover is set to prove that there exists a solution σ_0 to the SPE-NCRS problem, while Challenger wants to disprove this claim. However, the special role of player 0 in the original game $\mathcal{G}$ must be taken into account. To this end, we need to *split* Prover into a coalition of two Provers, *Prover* 1 and *Prover* 2 (we later show, in Section 4, how to come back to the two-player model). This game is called P_1CP_2 *game* and is denoted by $P_1CP_2(\mathcal{G})$.

Let us give some intuition. The three players P_1, C, and P_2, proceed to construct, step by step and with some additional interactions, a play in $P_1CP_2(\mathcal{G})$, that *simulates* a play in $\mathcal{G}$ (see Definition 15 below). Each player has a specific part to play in the construction of

this simulating play. As mentioned previously, player P_1 has the particular task to exhibit a candidate solution strategy σ_0 to the SPE-NCRS problem for $\mathcal{G}$. This is done by simulating player 0 playing σ_0 and not deviating from it. Thus, along the play in $P_1CP_2(\mathcal{G})$ being constructed, whenever the corresponding state in the simulated play of $\mathcal{G}$ belongs to player 0, it is then up to P_1 to choose the next state. One of the tasks of player C is to try to complete σ_0 with a subgame perfect response $\bar{\sigma}_{-0}$ from the other players in $\mathcal{G}$. Thus, whenever the corresponding state in the simulated play of $\mathcal{G}$ belongs to a player $i \neq 0$, it is player C's turn to play in $P_1CP_2(\mathcal{G})$: he *proposes* to play an action according to some strategy σ_i of player i in $\mathcal{G}$ and in addition *predicts* the gain profile of what will be the outcome of the profile $(\sigma_0, \bar{\sigma}_{-0})$, that is, of the simulated play being constructed. However, since players other than player 0 in $\mathcal{G}$ can deviate, whenever C has made a proposal for a next state, he has to let player P_2 have a say: P_2 can either accept this proposal or refuse it and *deviate* on behalf of player i by choosing another state as the next one in the simulating play. This phase is called the *decision phase*. If this phase results in a deviation, the game proceeds to the *adjusting* phase: since the current subgame has changed, Challenger has to predict the gain profile of $(\sigma_0, \bar{\sigma}_{-0})$ in this subgame, that shows the deviation of player i was not a profitable one. Then, the play in $P_1CP_2(\mathcal{G})$ resumes to the construction phase again.

How do we ensure that σ_0 is fixed? In other words, what restricts P_1 from adapting his strategy to the way C and P_2 interact, but only according to the so far constructed simulated history in G? The solution we choose is to make use of *imperfect information*: in the P_1CP_2 game, P_1 cannot, in fact, distinguish between all the states. To model the partial view P_1 has on the states of the game, we speak of *observations*. Informally, to each state (and action) is associated an observation, and P_1 has to take action upon sequences of observations only (that we call *observed histories*). In particular, it is assumed that the strategies should *respect* the observations, in the sense that two different histories yielding the same sequence of observations should trigger the same action of P_1. Thus, we only consider these *observation-based strategies* as possible behaviors of P_1. We now proceed to the formal definition of the P_1CP_2 game. In Section 3.4, we show that solving the SPE-NCRS problem is equivalent to solving this game in a sense stated in Theorem 20.

3.2 Observation Functions

We begin by introducing the concepts of observation and of observation-based strategy. Let $G = (V, A, \Pi, \delta, v_0)$ be a game structure and player i be some player in G. Let $\mathcal{O}$ be a partition of V, that defines an *observation function* $\mathcal{O}bs : V \to \mathcal{O}$ for player i, that is, for every state v, we have $\mathcal{O}bs(v) = o \in \mathcal{O}$. Let then $\bar{\mathcal{O}}$ be a partition of A, that extends the previous function $\mathcal{O}bs$ to actions, that is, for every action a, we have $\mathcal{O}bs(a) = \bar{o} \in \bar{\mathcal{O}}$. The function $\mathcal{O}bs$ extends to histories and plays in the straightforward way. We say that player i *observes* G *through* $\mathcal{O}bs$ if he cannot distinguish between states (resp. actions, histories, plays) that yield the same observation via the function $\mathcal{O}bs$. For those states or actions that are the unique element of their observation set, we say that they are *visible*. We say that $\mathcal{O}bs$ is the *identity function* if $\mathcal{O}bs(v) = \{v\}$ for all $v \in V$ and $\mathcal{O}bs(a) = \{a\}$ for all $a \in A$. Whenever player i observes the game structure through the identity function, we say that player i has *perfect information*. If it is through any other function, we say that he has *imperfect information*. We will always assume that a player *sees his own actions perfectly* (that is, his observation function restricted to the domain of his action set corresponds to the identity function). Furthermore, we assume all players have *perfect recall*: they remember the full sequence of observations they witnessed from the start of the play. In the sequel, we only consider *player-stable* observation functions such that the last states of the histories observed similarly all belong to the same player.

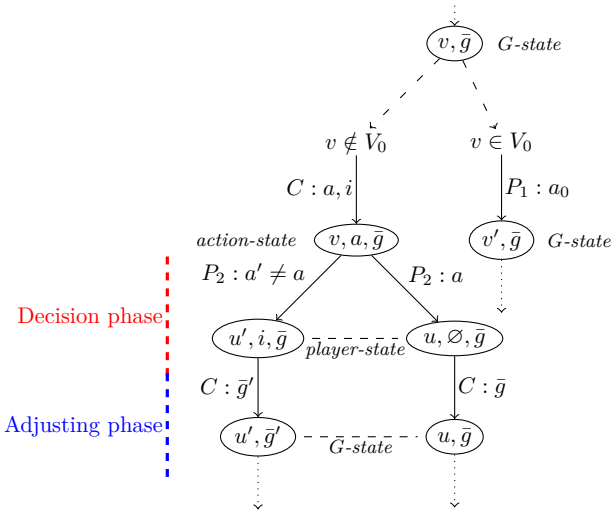


Figure 2 General structure of a play in $P_1CP_2(\mathcal{G})$.

▶ **Definition 12** (Player-stable observation function). *An observation function $\mathcal{O}bs$ for player i is* player-stable *if for every two histories $h = v_0a_0v_1a_1 \dots v_k$ and $g = u_0b_0u_1b_1 \dots u_k$ such that $\mathcal{O}bs(h) = \mathcal{O}bs(g)$, then the states v_ℓ, u_ℓ are controlled by the same player, for all $\ell \in \{0, \dots, k\}$.*

The identity observation function is trivially player-stable. A game $\mathcal{G}$ with objectives W_i for all its players $i \in \Pi$ and a player-stable observation function $\mathcal{O}bs$ for one of its players is denoted $\mathcal{G} = (G, (W_i)_{i\in\Pi}, \mathcal{O}bs)$. When $\mathcal{O}bs$ is the identity, we do not mention it in this notation. Notice that with a player-stable observation function for player i, we have that, given a history $h \in \mathsf{Hist}^i_G$, all histories h' with the same observation as h also belong to Hist^i_G. It is therefore natural to ask that a strategy σ_i for player i is *observation-based*, that is, the same action is played by player i after all histories observed similarly.

▶ **Definition 13** (Observation-based strategy). *Given a player-stable observation function $\mathcal{O}bs$, a strategy σ_i of player i is* observation-based *if for every pair of histories h, h', if $h \in \mathsf{Hist}^i_G$ and $\mathcal{O}bs(h) = \mathcal{O}bs(h')$, then $\sigma_i(h) = \sigma_i(h')$.*

Both concepts of observation and observation-based strategy are illustrated in an example in Appendix B.1.

3.3 Definition of the P_1CP_2 Game

In this section, let us fix a game $\mathcal{G} = (G, (W_i)_{i\in\Pi})$ with a game structure $G = (V, A, \Pi, \delta, v_0)$ and objectives $(W_i)_{i\in\Pi}$. Later, this game will be a reachability game or a parity game. We here formally define the corresponding P_1CP_2 game, that was informally presented in Section 3.1. Recall that the role of player P_1 is to simulate a strategy σ_0 of player 0 in $\mathcal{G}$, while player C has to propose a subgame perfect response to σ_0 such that $(\sigma_0, \bar{\sigma}_{-0})$ is losing for player 0 in $\mathcal{G}$, response to which player P_2 can react by choosing some deviations (decision phase). The interactions between the three players result in a play in $P_1CP_2(\mathcal{G})$ that simulates a play in $\mathcal{G}$. Challenger has to predict a gain profile for this simulated play, which he can update upon deviations from P_2 (adjusting phase). We begin by defining the

game structure of the new game $P_1CP_2(\mathcal{G})$.[2] We then define the observation function of P_1 in Definition 19; and we finally define and describe the objectives of the three players. Figure 2 should help the reader for understanding those definitions. The states (except its initial state v'_0) of the P_1CP_2 game structure have several components: All of them have a state of G as their first component, that we call *G-component*. Similarly, all of them have a *gain-component*, which consists of a gain profile $\bar{g} \in \{0,1\}^{|\Pi|}$ for the players of G, where in each component g_i of $\bar{g}$, 0 symbolizes a loss and 1 a win with respect to objective W_i. The states that have only these two components are called *G-states* and belong either to P_1 or C. A projection on the G-components of these states determines the simulated play in G (see Definition 15). As already stated in Section 3.1, whenever the current state of the simulated play belongs to player 0 in G, the next state of the simulated play is chosen directly by P_1. However, when it is not the case, C has to make a proposition, which should then be validated or changed by P_2 (decision phase). Thus, to reflect this proposal while remaining hidden to P_1, the successor states of G-states belonging to C have an extra component, called *action-component*, which records the action of G proposed by C. From such states called *action-states*, P_2 has to react by confirming or changing the action that was proposed by C. This is modeled by a third kind of states, called *player-states*. Such states have a *player-component*, which consists of a player from G different from player 0 to signal to C this player has indeed deviated (or the empty set, to signal acceptance by Prover 2 of Challenger's proposal).

▶ **Definition 14** (States of the P_1CP_2 game)**.** *Given a game $\mathcal{G}$, the game structure of $P_1CP_2(\mathcal{G})$ is a game structure $G' = (V', A', \{P_1, C, P_2\}, \delta', v'_0)$, where P_1, C, and P_2 are the three players and the set of states $V' = V'_{P_1} \cup V'_C \cup V'_{P_2}$ is as follows:*

- $V'_{P_1} = \{(v, \bar{g}) \mid v \in V_0,\ \bar{g} \in \{0,1\}^{|\Pi|}\}$,
- $V'_C = \{v'_0\} \cup \{(v, \bar{g}) \mid v \in V \setminus V_0,\ \bar{g} \in \{0,1\}^{|\Pi|}\} \cup \{(v, i, \bar{g}) \mid v \in V,\ i \in \Pi \setminus \{0\} \cup \{\varnothing\},\ \bar{g} \in \{0,1\}^{|\Pi|}\}$,
- $V'_{P_2} = \{(v, a, \bar{g}) \mid v \in V \setminus V_0,\ a \in A \setminus A_0$ *and* $\delta(v, a)$ *is defined,* $\bar{g} \in \{0,1\}^{|\Pi|}\}$,
- v'_0 *is the initial state.*

Among those states, $(v, \bar{g})$ are G-states, $(v, a, \bar{g})$ are action-states, and $(v, i, \bar{g})$ are player-states. Moreover, v is a G-component, $\bar{g}$ is a gain-component, a is an action-component, and i is a player-component. Notice that the sets V'_{P_1}, V'_C, V'_{P_2} are pairwise disjoint.

Let us now describe the transitions of the P_1CP_2 game and explain further the actions of the players, see also Figure 2 . The goal of Challenger is to prove the existence of a σ_0-fixed SPE losing for player 0 in $\mathcal{G}$. To be able to verify this claim in $P_1CP_2(\mathcal{G})$, Challenger has to predict the gain of the simulated play that is being constructed. To do so, the gain-components of the states of $P_1CP_2(\mathcal{G})$ are used as follows: First, Challenger owns the initial state v'_0, and has to choose a gain profile $\bar{g}$, losing for player 0,[3] to start the construction of the simulated play in the G-state $(v_0, \bar{g})$. Then, whenever P_2 chooses to make some player i deviate, before reaching a G-state, Challenger has to respond by choosing an adjusted gain profile with a lower or equal gain for this player i (adjusting phase). This new gain shows to P_2 the absence of any profitable deviation for player i. Note that for the P_1CP_2 game structure, two modeling decisions have been made to help handle the subsequent

[2] Note that to avoid confusion with the players of G, and without loss of generality, the three players of the P_1CP_2 game have been given explicit names rather than numbers.

[3] As an outcome of a subgame perfect response to σ_0 losing for player 0 is exactly what Challenger wants to exhibit, for any σ_0 simulated by P_1.

developments. First, to ensure a certain regularity of the plays' shape in the P_1CP_2 game and that P_1 cannot infer whether P_2 actually made a deviation, the adjusting phase is played regardless of a deviation occurring or not: if there was no deviation, Challenger has no choice but to play the same gain profile. Second, to avoid confusion on who, among Challenger and Prover 2, is currently playing some action $a \in A \setminus A_0$, this action is of the form (a, i) for C and a for P_2, that is, we additionally recall the player i from G performing the action a when Challenger is playing.

As already mentioned in Section 3.1, given a play in $P_1CP_2(\mathcal{G})$, there exists a unique corresponding play in G being simulated by the interactions of the Provers and Challenger.

▶ **Definition 15** (Simulated play and gain in G)**.** *Let $\rho = v'_0 a'_0 v'_1 a'_1 v'_2 a'_2 \dots$ be a play in the game structure of $P_1CP_2(\mathcal{G})$. The* simulated play *of ρ is the play $\mathsf{sim}(\rho) \in \mathsf{Plays}_G$ being the projection of ρ on the G-component of G-states (which belong either to P_1 or C) and on actions of P_1 and P_2.*[4] *The definition extends naturally to histories. The* simulated gain *of ρ is a Boolean vector $\mathsf{simGain}(\rho) \in \{0,1\}^{|\Pi|}$, such that $\mathsf{simGain}(\rho)_i = 0$ if $\mathsf{sim}(\rho)$ is losing for player i, and $\mathsf{simGain}(\rho)_i = 1$ if $\mathsf{sim}(\rho)$ is winning for player i.*

Let us give an example:

▶ **Example 16.** Consider the fictional history $h = \ v'_0\ \bar{g}\ (v_0, \bar{g})\ a_0\ (v_1, \bar{g})\ (a_1, i)\ (v_1, a_1, \bar{g})\ a'_1\ (v_2, i, \bar{g})\ \bar{g}'\ (v_2, \bar{g}')$ in the P_1CP_2 game of some game $\mathcal{G}$. It starts in the initial state v'_0, where C chooses a gain profile $\bar{g}$ where $g_0 = 0$. Thus the second state of the history is the G-state $(v_0, \bar{g})$. Looking at the form of the third state $(v_1, \bar{g})$ in h, we can deduce that $(v_0, \bar{g})$ belongs to Prover 1, that is, v_0 belongs to player 0 in G. Moreover $\delta(v_0, a_0) = v_1$. Then, the fourth state $(v_1, a_1, \bar{g})$ in h is an action-state, which means that the previous state $(v_1, \bar{g})$ belongs to Challenger, such that v_1 belongs to player $i \neq 0$ in G, and a_1 is the action from G proposed by C in this scenario. One can see that P_2 then chooses the action a'_1, which is different from a_1 since the fifth state $(v_2, i, \bar{g})$ of h is a state whose player-component i is not equal to $\varnothing$. This indicates that P_2 makes player i deviate in G and that $\delta(v_1, a'_1) = v_2$. By now, Challenger has to choose a new gain profile $\bar{g}'$, yielding the last state $(v_2, \bar{g}')$ of h, which is a G-state. By projecting h on the G-components of its G-states and on the actions of the Provers, one can check that the simulated history of h in G is $\mathsf{sim}(h) = v_0 a_0 v_1 a'_1 v_2$. Consider now the fictional play $\rho = h\ \big((a_2, j)\ (v_2, a_2, \bar{g}')\ a_2\ (v_2, \varnothing, \bar{g}')\ \bar{g}'\ (v_2, \bar{g}')\big)^\omega$ in the P_1CP_2 game. Looking at ρ, one can deduce that from $(v_2, \bar{g}')$, the last state of h, Challenger proposes action a_2 from G for player $j \neq 0$ and that P_2 accepts this proposal. Furthermore, Challenger cannot adjust the gain profile, leading to state $(v_2, \bar{g}')$. One can see that this behavior repeats indefinitely, thus the corresponding simulated play is $\mathsf{sim}(\rho) = \mathsf{sim}(h)(a_2 v_2)^\omega$. The simulated gain $\mathsf{simGain}(\rho)$ of ρ is a Boolean vector deduced from $\mathsf{sim}(\rho)$, such that its i-th component is equal to 1 if, and only if, $\mathsf{sim}(\rho)$ is winning for player i.

▶ Remark 17. Note that while there exists a unique simulated play $\mathsf{sim}(\rho)$ in $\mathcal{G}$ for every play ρ in $P_1CP_2(\mathcal{G})$, the converse does not hold. Indeed, several different sequences of interactions between P_2 and C yield the same simulated play in $\mathcal{G}$. For instance, Challenger can propose different actions that P_2 can refuse.

We have seen that each state and action of G appear in several contexts in the game structure G' of $P_1CP_2(\mathcal{G})$. Let us now state how the size of G' depends on the size of G. The proof of this lemma directly follows from the definition of the P_1CP_2 game.

[4] This projection does not take into account the action- and player-states and the actions of Challenger.

▶ **Lemma 18.** *Given the size $|V|$, $|A|$, and $|\Pi|$ of the game structure of $\mathcal{G}$, the game structure of $P_1CP_2(\mathcal{G})$ has 3 players and a size $|V'|$ linear in $|V|$ and $|A|$, and exponential in $|\Pi|$, and $|A'|$ linear in $|A|$ and exponential in $|\Pi|$.*

In the P_1CP_2 game, recall that C and P_2 have perfect information whereas P_1 has imperfect information to ensure that P_1 cannot adapt his strategy from observing the interactions between C and P_2. The observation function $\mathcal{O}bs$ for player P_1 is defined below on V' and A'. For each state v' of V', he is only able to observe the G-component of v'. Concerning the actions of A', those of $A'_{P_1} \cup A'_{P_2}$ are all visible whereas those of A'_C are not visible at all. In the next definition, there is an abuse of notation in $\mathcal{O}$ and $\bar{\mathcal{O}}$.

▶ **Definition 19** (Information in the P_1CP_2 game)**.** *Given the game structure G' of $P_1CP_2(\mathcal{G})$, players C and P_2 have perfect information and the observation function $\mathcal{O}bs$ of player P_1 is defined as follows: We have $\mathcal{O}bs : V' \to \mathcal{O} = \{v'_0\} \cup \{v \mid v \in V\}$ such that $\mathcal{O}bs(v, \bar{g}) = v$ for all $(v, \bar{g}) \in V'_{P_1}$, $\mathcal{O}bs(v'_0) = v'_0$, $\mathcal{O}bs(v, \bar{g}) = v$ for all $(v, \bar{g}) \in V'_C$, $\mathcal{O}bs(v, i, \bar{g}) = v$ for all $(v, i, \bar{g}) \in V'_C$, and $\mathcal{O}bs(v, a, \bar{g}) = v$ for all $(v, a, \bar{g}) \in V'_{P_2}$. Furthermore, we have $\mathcal{O}bs : A' \to \bar{\mathcal{O}} = \{\sharp\} \cup \{a \mid a \in A\}$ such that $\mathcal{O}bs(a') = a'$ for all $a' \in A'_{P_1}$, $\mathcal{O}bs(a') = \sharp$ for all $a' \in A'_C$, and $\mathcal{O}bs(a') = a'$ for all $a' \in A'_{P_2}$.*

To finalize the definition of the P_1CP_2 game, it remains to define the objectives of the three players. Recall that the objective of Challenger is to show that for each strategy σ_0 of player 0 in $\mathcal{G}$, there exists a subgame perfect response $\bar{\sigma}_{-0}$ such that the outcome of $(\sigma_0, \bar{\sigma}_{-0})$ is losing for player 0. Let us give some intuition on what is a play ρ in the P_1CP_2 game that is winning for C. Three winning situations may occur:

(*iC*) Eventually, that is, after reaching some subgame, P_2 always accepts the action proposals of C and the gain predicted by C in the subgame is correct (it is equal to the simulated gain in this subgame), or

(*iiC*) eventually, P_2 keeps making one unique player i deviate in the decision phase, but C is able to adjust the gain to show that this deviation is not profitable for player i, or finally

(*iiiC*) P_2 keeps making at least two different players deviate, essentially conceding the play, as the only deviations that can be considered within the scope of (0-fixed) SPEs are *unilateral* deviations.

In the P_1CP_2 game, the two Provers have the same objective W_P that is opposed to the objective W_C of Challenger. Indeed, recall that their objective is to exhibit a strategy σ_0 for player 0 in G, such that for every subgame-perfect response $\bar{\sigma}_{-0}$ to σ_0, the outcome of the resulting profile $(\sigma_0, \bar{\sigma}_{-0})$ is winning for player 0. Let us give some intuition on what is a winning play for the Provers. Two winning situations may occur for the Provers along a play ρ in the P_1CP_2 game:

(*iP*) Eventually, after reaching some subgame, P_2 always accepts the action proposals of C and the gain predicted by C is incorrect, or

(*iiP*) eventually, P_2 keeps making one unique player i deviate in the decision phase, and C is not able to adjust the gain to show that this deviation is not profitable for player i.

A full formal definition of the players' objectives is provided in Appendix B.2.

Given a game $\mathcal{G}$, we have $P_1CP_2(\mathcal{G}) = (G', W_P, \mathcal{O}bs)$ its P_1CP_2 game, where W_P is the objective of the Provers.

3.4 Equivalence Between SPE-NCRS Problem and P_1CP_2 Game

In this section, we show that solving the SPE-NCRS problem for a game $\mathcal{G}$ is equivalent to solving its P_1CP_2 game, as stated in the following theorem. To keep things short, we give here a sketch of the proof, and refer the reader to [19] for a full and detailed account.

▶ **Theorem 20** (Equivalence theorem between $\mathcal{G}$ and $P_1CP_2(\mathcal{G})$)**.** *Let $\mathcal{G} = (G, (W_i)_{i\in\Pi})$ be a game, and $P_1CP_2(\mathcal{G}) = (G', W_P, \mathcal{O}bs)$ be its P_1CP_2 game. There exists in $\mathcal{G}$ a strategy σ_0 of player 0, such that, for every subgame-perfect response $\bar{\sigma}_{-0}$, the play $\langle(\sigma_0, \bar{\sigma}_{-0})\rangle_{v_0}$ is winning for player 0 if, and only if, there exists in $P_1CP_2(\mathcal{G})$ an observation-based strategy τ_{P_1} of P_1 such that for all strategies τ_C of C, there exists a strategy τ_{P_2} of P_2 such that the play $\langle\tau_{P_1}, \tau_C, \tau_{P_2}\rangle_{v'_0}$ belongs to W_P.*

Proof sketch. Let σ_0 be a solution to the SPE-NCRS problem of $\mathcal{G}$. Let τ_{P_1} be the observation-based strategy of P_1 that simulates σ_0: it is naturally defined as the strategy that makes the same choice of action as σ_0 at h in $\mathcal{G}$ at all the histories that are observed as h for P_1 in $P_1CP_2(\mathcal{G})$. Let τ_C be a strategy of C. Let $\bar{\sigma}_{-0}$ be the strategy profile obtained from τ_C by extracting the actions chosen by C at histories that indeed simulate histories in $\mathcal{G}$. There are two possibilities, either $(\sigma_0, \bar{\sigma}_{-0})$ is a 0-fixed SPE in $\mathcal{G}$, or it is not a 0-fixed SPE.

1. Suppose $(\sigma_0, \bar{\sigma}_{-0})$ is a 0-fixed SPE in $\mathcal{G}$. Let τ_{P_2} be τ_{acc}, the *accepting strategy* of P_2, where he accepts every action proposal of C and never deviates. By definition of τ_{P_2}, the simulated play of $\langle\tau_{P_1}, \tau_C, \tau_{P_2}\rangle_{v'_0}$ is $\langle\sigma_0, \bar{\sigma}_{-0}\rangle_{v_0}$. In that case, since σ_0 is a solution to the SPE-NCRS problem of $\mathcal{G}$, we know that $\langle\sigma_0, \bar{\sigma}_{-0}\rangle_{v_0} \in W_0$. However, by construction of $P_1CP_2(\mathcal{G})$ and the fact that τ_{P_2} is the accepting strategy of P_2, we know that C has predicted a loss for player 0 from the start, and has never had to adjust his prediction. Thus, we are in the case (iP) of the winning condition for the Provers.
2. Suppose now $(\sigma_0, \bar{\sigma}_{-0})$ is not a 0-fixed SPE in $\mathcal{G}$. This means that it is not a 0-fixed NE in some subgame of $\mathcal{G}$. Let h be a history compatible with σ_0 such that $(\sigma_0, \bar{\sigma}_{-0})_{\restriction h}$ is not an NE. Let player $i \neq 0$ be a player in $\mathcal{G}$ that has a profitable deviation σ'_i in the subgame $\mathcal{G}_{\restriction h}$ starting after h. In $P_1CP_2(\mathcal{G})$, we can show that there exists a unique history h' ending in a G-state, compatible with τ_C, such that $\mathsf{sim}(h') = h$ and that is also compatible with τ_{P_1}. Depending on Challenger's prediction for player i at h', the strategy τ_{P_2} has to be different: if Challenger predicted a loss, we let τ_{P_2} bring the play in h', then simulate the profitable deviation of player i. If Challenger predicted a win, we let τ_{P_2} bring the play in h', then switch to τ_{acc}: since player i has a profitable deviation from h', it is necessary the case that the resulting play in $\mathcal{G}$ is losing for player i.

Turning to the other direction, assume that there is no solution to the SPE-NCRS problem in $\mathcal{G}$. Let τ_{P_1} be a strategy of P_1 in $P_1CP_2(\mathcal{G})$. Let σ_0 be the strategy of player 0 in $\mathcal{G}$ such that τ_{P_1} is its simulation. By Theorem 9, there exists a σ_0-fixed SPE in $\mathcal{G}$. Among all those σ_0-fixed SPEs, one must have an outcome losing for player 0 as, by assumption, σ_0 is not a solution of the SPE-NCRS problem in $\mathcal{G}$. Let $\bar{\sigma} = (\sigma_0, \bar{\sigma}_{-0})$ be such a σ_0-fixed SPE. From these, we can define a strategy τ_C for C that will precisely enact $\bar{\sigma}_{-0}$ on relevant histories, and predict correctly the gain profile as the one σ in every subgame. In particular, at the initial state, it predicts the gain profile $\bar{g}$ of $\bar{\sigma}$, such that $g_0 = 0$ by choice of the 0-fixed SPE.

Let now τ_{P_2} be a strategy of P_2. Consider the play $\rho = \langle\tau_{P_1}, \tau_C, \tau_{P_2}\rangle_{v'_0}$. If there are infinitely many deviations by two different players of $\mathcal{G}$ in the play ρ, then ρ satisfies the winning condition $(iiiC)$ of Challenger. Assume now that there is at most one player that P_2 makes deviate. There are two cases:

1. Either τ_{P_2} prescribes a finite (possibly null) number of deviations along ρ, and then switches to the accepting strategy τ_{acc}. Eventually, after reaching some subgame, P_2 always accepts the action proposals of C and the gain predicted by C in the subgame is correct. Hence ρ satisfies the winning condition (iC) of Challenger.
2. Or τ_{P_2} prescribes an infinite number of deviations for the same player i along ρ. As there are only two values (0 and 1) for the gain-components and there is no opportunity for the i-component of the predicted gain to increase, we eventually get a stable gain-component

g_i for player i. If $g_i = 1$, we clearly have $g_i = 1 \geq \mathsf{simGain}(\rho)_i$, showing that (iiC) is satisfied. If $g_i = 0$, recall that $\bar{\sigma}$ is a 0-fixed SPE, thus it is an NE in the subgame from which g_i was stabilized. The outcome of $\bar{\sigma}$ in this subgame is losing for player i as Challenger correctly predicts a gain of 0 with τ_C. Any deviation from this outcome is thus also losing for player i, including $\mathsf{sim}(\rho)$. Thus, (iiC) is again satisfied. ◀

4 Solving the P_1CP_2 Game

This section is the last puzzle piece to complete the proof of Theorem 11. We present here the key ideas behind the manyfold process of solving the P_1CP_2 game, while the full technical constructions and details can be found in [19]. Thanks to Theorem 20, we have an equivalence between the existence of a solution of the SPE-NCRS problem for a game and the fact that the Provers are able to win the associated P_1CP_2 game. More precisely, the situation where P_1 has *a* strategy, such that for *every* strategy of Challenger, P_2 has *a* strategy to make the Provers win. The remaining question is how to determine whether this is the case? In other words, how can one *solve*, in this particular sense, this three-player game with imperfect information? As shown on Figure 1 , this involves several steps. We start by a first technical step to obtain a P_1CP_2 game with a *Rabin objective* for the Provers. Then, we get rid of the three-player setting, by eliminating one Prover to obtain a two-player PC game with imperfect information. Once we have such a PC game, we focus on getting rid of the imperfect information and work with a parity objective instead of a Rabin one. This allows us to obtain an equivalent *two-player zero-sum parity game with perfect information*, which is effectively solvable. The techniques used are similar to what can be found in the literature ([21, 22, 41, 42]). However, their settings each differ slightly from the others and ours. Thus, to obtain finer complexity measures, we adapt their techniques to our setting.

The P_1CP_2 Game as a Rabin Game. Given a parity game $\mathcal{G}$, we show that its corresponding P_1CP_2 game can be seen as a three-player game with the objective W_P for the two Provers translated into a *Rabin* objective. The approach is to use a deterministic automaton $\mathcal{O}$ that *observes* the states of $P_1CP_2(\mathcal{G})$. Then the *synchronized product* of the game structure of $P_1CP_2(\mathcal{G})$ with this observer automaton $\mathcal{O}$ is equipped with a Rabin objective translating W_P and thus leads to the announced three-player Rabin game.

From two Provers to one Prover. The next step is to merge P_1 and P_2 into a unique Prover by using a technique inspired from [22]. The main idea is to use imperfect information to ensure merging the two Provers does not grant too much knowledge to the new single Prover. Indeed, if the new Prover had perfect information, he could not simulate Prover 1 truthfully. Thus we let the new Prover have the same level of information as Prover 1 in the P_1CP_2 game. However, in order to let the new Prover have as much actions available as Prover 2 and stay observation-based, we modify the action set to include all *functions* from states of P_2 to actions of P_2. This way, the "merged" Prover preselects actions for each possible Challenger move, effectively encoding both Provers' strategies via a state-to-action function. This yields an exponential blowup in the number of actions. The Rabin objective remains the same. We obtain the corresponding PC game, denoted $PC(\mathcal{G})$.

Eliminating Imperfect Information. To solve the two-player game $PC(\mathcal{G})$, we get rid of the imperfect information and then apply standard game-theoretic techniques. In the literature [21, 41, 42], this is usually done in two steps: first, make the objective *visible*, that is, such

that any two similarly observed plays agree on the winning condition, second, apply the *subset construction* to recall the set of possible visited states, and letting them be observed. This can be done in only one step, by simultaneously modifying the game structure to both entail the subset construction on the states of the PC game, and the product with an automaton that monitors the winning condition along the plays. This allows us to limit the exponential blowups to only one in the size of the state space and the Rabin condition. Essentially, the monitoring automaton is a deterministic parity automaton that is obtained by complementing a non-deterministic Streett-automaton that detects sets of similarly observable plays that contain a losing play for the now unique Prover.

Complexity. Recall that we set out from a parity game $\mathcal{G} = (G, (W_i)_{i\in\Pi})$ with $|V|$, $|A|$, and $|\Pi|$ be the size of the game structure G, and $|\alpha_i|$ be the size of each parity condition W_i. By the construction mentioned above, we have an equivalence with a Rabin P_1CP_2 game with a state set of size linear in $|V|$ and $|A|$, and exponential in $|\Pi|$, an action set linear in $|A|$ and exponential in $|\Pi|$, while its Rabin objective has a size linear in $|\Pi|$ and the $|\alpha_i|$. Reducing to only one Prover, we obtain an exponential number of actions compared to the two-Provers version, thus exponential in $|V|$, $|A|$ and double-exponential in $|\Pi|$. As mentioned above, with the removal of imperfect information, we obtain a game with a size of the state set that is now exponential in $|V|$, $|A|$, the $|\alpha_i|$ and double exponential in $|\Pi|$, while the new parity condition is polynomial in $|V|, |A|, |\alpha_i|, \forall i$ and exponential in $|\Pi|$. Finally, by [20], solving the ultimate parity game, or equivalently the SPE-NCRS problem, is in time exponential in $|V|, |A|, |\alpha_i|, \forall i$, and double-exponential in $|\Pi|$. As the game structure of $\mathcal{G}_0$ is action-unique (see Definition 1), it follows that $|A| \leq |V|^2$, thus leading to an algorithm in time exponential in $|V|$ and each $|\alpha_i|$, and double-exponential in $|\Pi|$. This completes the proof of the complexity upper bound of Theorem 11. For the PSpace lower bound, the QBF-reduction of [26] (Theorem 7) applies here; all NE responses to σ_0 in that proof are also SPEs. For the complexity lower bound of Theorem 11 when the number of players is fixed, we adapt the reduction proposed for the NE-NCRS problem in [26], in a way to deal with SPEs instead of NEs.

The Case of Reachability. A careful analysis of the simpler case of reachability games, when *the number of players is fixed*, leads to a more fine-tuned solution and a better complexity result: Theorem 11 states a *polynomial complexity* instead of the exponential complexity of the parity case. This approach follows the same steps as presented above for parity games, but with a few adjustments. The key idea is that monitoring reachability objectives is simpler than parity ones, which means that the synchronized product with an observer automaton $\mathcal{O}$ can omit some information from the original game structure. Indeed, when monitoring a play in the P_1CP_2 game that stems from an original reachability game, one still needs to keep track of the actions of both Provers and Challenger, but the checking of the gain component is simplified compared to parity: only the information of whether each player has already visited his target set is to be remembered. This can be exploited to avoid the exponential blowup of the state space in the subset construction phase (with respect to the original state set V) to get rid of imperfect information and thus obtain a better complexity.

▶ Remark 21. Suppose that our algorithm establishes the existence of a solution σ_0 to the SPE-NCRS problem for parity games. As it is obtained from a memoryless winning strategy in the *final* parity zero-sum game, we get a finite-memory solution σ_0 whose memory size is exponential when the number of players is fixed, doubly exponential otherwise. For a lower bound on the memory required, it is fairly straightforward to show that it requires exponential memory by reducing a Streett zero-sum game to our problem.

5 Conclusion

In this work, we introduce a novel algorithm to solve the SPE-NCRS problem for parity objectives. Unlike previous methods that converted the problem into a model-checking problem for Strategy Logic, our algorithm reduces the SPE-NCRS problem to a three-player zero-sum game with imperfect information, framed as a Prover-Challenger game. This new angle yields improved complexity upper bounds: exponential time in the number of vertices of the game structure and the number of priorities of the parity objectives, doubly exponential time in the number of players. In particular, our algorithm runs within exponential time for a fixed number of players, which is particularly relevant since the number of players is typically small in practical scenarios. Moreover, we establish a lower bound that indicates the impossibility of solving the SPE-NCRS problem in polynomial time unless $\mathsf{P} = \mathsf{NP}$ even for a fixed number of players. For the particular case of reachability objectives, when the number of players is fixed, we prove polynomial complexity like for the NE-NCRS problem [26]. We believe that the Prover-Challenger framework, based on a three-player model with imperfect information, may be applicable for other synthesis challenges beyond our current application.

References

1 Rajeev Alur, Thomas A. Henzinger, Orna Kupferman, and Moshe Y. Vardi. Alternating refinement relations. In Davide Sangiorgi and Robert de Simone, editors, *CONCUR '98: Concurrency Theory, 9th International Conference, Nice, France, September 8-11, 1998, Proceedings*, volume 1466 of *Lecture Notes in Computer Science*, pages 163–178. Springer, 1998. doi:10.1007/BFB0055622.

2 Roderick Bloem, Krishnendu Chatterjee, and Barbara Jobstmann. Graph games and reactive synthesis. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 921–962. Springer, 2018. doi:10.1007/978-3-319-10575-8_27.

3 Patricia Bouyer, Romain Brenguier, Nicolas Markey, and Michael Ummels. Pure Nash equilibria in concurrent deterministic games. *Log. Methods Comput. Sci.*, 11(2), 2015. doi:10.2168/LMCS-11(2:9)2015.

4 Romain Brenguier, Lorenzo Clemente, Paul Hunter, Guillermo A. Pérez, Mickael Randour, Jean-François Raskin, Ocan Sankur, and Mathieu Sassolas. Non-zero sum games for reactive synthesis. In Adrian-Horia Dediu, Jan Janousek, Carlos Martín-Vide, and Bianca Truthe, editors, *Language and Automata Theory and Applications - 10th International Conference, LATA 2016, Prague, Czech Republic, March 14-18, 2016, Proceedings*, volume 9618 of *Lecture Notes in Computer Science*, pages 3–23. Springer, 2016. doi:10.1007/978-3-319-30000-9_1.

5 Léonard Brice, Jean-François Raskin, and Marie van den Bogaard. The complexity of SPEs in mean-payoff games. In Mikolaj Bojanczyk, Emanuela Merelli, and David P. Woodruff, editors, *49th International Colloquium on Automata, Languages, and Programming, ICALP 2022, July 4-8, 2022, Paris, France*, volume 229 of *LIPIcs*, pages 116:1–116:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICS.ICALP.2022.116.

6 Léonard Brice, Jean-François Raskin, and Marie van den Bogaard. On the complexity of SPEs in parity games. In Florin Manea and Alex Simpson, editors, *30th EACSL Annual Conference on Computer Science Logic, CSL 2022, February 14-19, 2022, Göttingen, Germany (Virtual Conference)*, volume 216 of *LIPIcs*, pages 10:1–10:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICS.CSL.2022.10.

7 Léonard Brice, Jean-François Raskin, and Marie van den Bogaard. Rational Verification for Nash and Subgame-Perfect Equilibria in Graph Games. In Jérôme Leroux, Sylvain Lombardy, and David Peleg, editors, *48th International Symposium on Mathematical Foundations of*

Computer Science, MFCS 2023, August 28 to September 1, 2023, Bordeaux, France, volume 272 of *LIPIcs*, pages 26:1–26:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICS.MFCS.2023.26.

8 Thomas Brihaye, Véronique Bruyère, Aline Goeminne, and Jean-François Raskin. Constrained existence problem for weak subgame perfect equilibria with ω-regular Boolean objectives. *Inf. Comput.*, 278:104594, 2021. doi:10.1016/J.IC.2020.104594.

9 Thomas Brihaye, Véronique Bruyère, Aline Goeminne, Jean-François Raskin, and Marie van den Bogaard. The complexity of subgame perfect equilibria in quantitative reachability games. *Log. Methods Comput. Sci.*, 16(4), 2020. URL: https://lmcs.episciences.org/6883.

10 Thomas Brihaye, Véronique Bruyère, Noémie Meunier, and Jean-François Raskin. Weak subgame perfect equilibria and their application to quantitative reachability. In Stephan Kreutzer, editor, *24th EACSL Annual Conference on Computer Science Logic, CSL 2015, September 7-10, 2015, Berlin, Germany*, volume 41 of *LIPIcs*, pages 504–518. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2015. doi:10.4230/LIPICS.CSL.2015.504.

11 Thomas Brihaye, Véronique Bruyère, and Gaspard Reghem. Quantitative Reachability Stackelberg-Pareto Synthesis is NEXPTIME-Complete. In Olivier Bournez, Enrico Formenti, and Igor Potapov, editors, *Reachability Problems - 17th International Conference, RP 2023, Nice, France, October 11-13, 2023, Proceedings*, volume 14235 of *Lecture Notes in Computer Science*, pages 70–84. Springer, 2023. doi:10.1007/978-3-031-45286-4_6.

12 Thomas Brihaye, Julie De Pril, and Sven Schewe. Multiplayer cost games with simple Nash equilibria. In Sergei N. Artëmov and Anil Nerode, editors, *Logical Foundations of Computer Science, International Symposium, LFCS 2013, San Diego, CA, USA, January 6-8, 2013. Proceedings*, volume 7734 of *Lecture Notes in Computer Science*, pages 59–73. Springer, 2013. doi:10.1007/978-3-642-35722-0_5.

13 Véronique Bruyère. Computer aided synthesis: A game-theoretic approach. In Émilie Charlier, Julien Leroy, and Michel Rigo, editors, *Developments in Language Theory - 21st International Conference, DLT 2017, Liège, Belgium, August 7-11, 2017, Proceedings*, volume 10396 of *Lecture Notes in Computer Science*, pages 3–35. Springer, 2017. doi:10.1007/978-3-319-62809-7_1.

14 Véronique Bruyère. Synthesis of equilibria in infinite-duration games on graphs. *ACM SIGLOG News*, 8(2):4–29, 2021. doi:10.1145/3467001.3467003.

15 Véronique Bruyère, Baptiste Fievet, Jean-François Raskin, and Clément Tamines. Stackelberg-Pareto synthesis. *ACM Trans. Comput. Log.*, 25(2):14:1–14:49, 2024. doi:10.1145/3651162.

16 Véronique Bruyère, Christophe Grandmont, and Jean-François Raskin. As soon as possible but rationally. In Rupak Majumdar and Alexandra Silva, editors, *35th International Conference on Concurrency Theory, CONCUR 2024, September 9-13, 2024, Calgary, Canada*, volume 311 of *LIPIcs*, pages 14:1–14:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPICS.CONCUR.2024.14.

17 Véronique Bruyère, Jean-François Raskin, and Clément Tamines. Pareto-Rational Verification. In Bartek Klin, Slawomir Lasota, and Anca Muscholl, editors, *33rd International Conference on Concurrency Theory, CONCUR 2022, September 12-16, 2022, Warsaw, Poland*, volume 243 of *LIPIcs*, pages 33:1–33:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.CONCUR.2022.33.

18 Véronique Bruyère, Stéphane Le Roux, Arno Pauly, and Jean-François Raskin. On the existence of weak subgame perfect equilibria. *Inf. Comput.*, 276:104553, 2021. doi:10.1016/J.IC.2020.104553.

19 Véronique Bruyère, Jean-François Raskin, Alexis Reynouard, and Marie Van Den Bogaard. The non-cooperative rational synthesis problem for subgame perfect equilibria and omega-regular objectives, 2024. doi:10.48550/arXiv.2412.08547.

20 Cristian S. Calude, Sanjay Jain, Bakhadyr Khoussainov, Wei Li, and Frank Stephan. Deciding parity games in quasi-polynomial time. *SIAM J. Comput.*, 51(2):17–152, 2022. doi:10.1137/17M1145288.

21 Krishnendu Chatterjee and Laurent Doyen. The complexity of partial-observation parity games. In Christian G. Fermüller and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 17th International Conference, LPAR-17, Yogyakarta, Indonesia, October 10-15, 2010. Proceedings*, volume 6397 of *Lecture Notes in Computer Science*, pages 1–14. Springer, 2010. doi:10.1007/978-3-642-16242-8_1.

22 Krishnendu Chatterjee and Laurent Doyen. Games with a weak adversary. In Javier Esparza, Pierre Fraigniaud, Thore Husfeldt, and Elias Koutsoupias, editors, *Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part II*, volume 8573 of *Lecture Notes in Computer Science*, pages 110–121. Springer, 2014. doi:10.1007/978-3-662-43951-7_10.

23 Krishnendu Chatterjee, Laurent Doyen, Emmanuel Filiot, and Jean-François Raskin. Doomsday equilibria for omega-regular games. *Inf. Comput.*, 254:296–315, 2017. doi:10.1016/J.IC.2016.10.012.

24 Krishnendu Chatterjee and Thomas A. Henzinger. Assume-guarantee synthesis. In Orna Grumberg and Michael Huth, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007, Proceedings*, volume 4424 of *Lecture Notes in Computer Science*, pages 261–275. Springer, 2007. doi:10.1007/978-3-540-71209-1_21.

25 Krishnendu Chatterjee, Thomas A. Henzinger, and Marcin Jurdzinski. Games with secure equilibria. *Theor. Comput. Sci.*, 365(1-2):67–82, 2006. doi:10.1016/J.TCS.2006.07.032.

26 Rodica Condurache, Emmanuel Filiot, Raffaella Gentilini, and Jean-François Raskin. The complexity of rational synthesis. In Ioannis Chatzigiannakis, Michael Mitzenmacher, Yuval Rabani, and Davide Sangiorgi, editors, *43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016, July 11-15, 2016, Rome, Italy*, volume 55 of *LIPIcs*, pages 121:1–121:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPICS.ICALP.2016.121.

27 Rodica Condurache, Youssouf Oualhadj, and Nicolas Troquard. The complexity of rational synthesis for concurrent games. In Sven Schewe and Lijun Zhang, editors, *29th International Conference on Concurrency Theory, CONCUR 2018, September 4-7, 2018, Beijing, China*, volume 118 of *LIPIcs*, pages 38:1–38:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPICS.CONCUR.2018.38.

28 Emmanuel Filiot, Raffaella Gentilini, and Jean-François Raskin. The Adversarial Stackelberg Value in Quantitative Games. In Artur Czumaj, Anuj Dawar, and Emanuela Merelli, editors, *47th International Colloquium on Automata, Languages, and Programming (ICALP 2020)*, volume 168 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 127:1–127:18, Dagstuhl, Germany, 2020. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ICALP.2020.127.

29 Dana Fisman, Orna Kupferman, and Yoad Lustig. Rational synthesis. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6015 of *Lecture Notes in Computer Science*, pages 190–204. Springer, 2010. doi:10.1007/978-3-642-12002-2_16.

30 Julian Gutierrez, Muhammad Najib, Giuseppe Perelli, and Michael J. Wooldridge. Automated temporal equilibrium analysis: Verification and synthesis of multi-player games. *Artif. Intell.*, 287:103353, 2020. doi:10.1016/J.ARTINT.2020.103353.

31 Julian Gutierrez, Muhammad Najib, Giuseppe Perelli, and Michael J. Wooldridge. On the complexity of rational verification. *Ann. Math. Artif. Intell.*, 91(4):409–430, 2023. doi:10.1007/S10472-022-09804-3.

32 Thomas A. Henzinger, Orna Kupferman, and Sriram K. Rajamani. Fair simulation. In Antoni W. Mazurkiewicz and Józef Winkowski, editors, *CONCUR '97: Concurrency Theory, 8th*

International Conference, Warsaw, Poland, July 1-4, 1997, Proceedings, volume 1243 of *Lecture Notes in Computer Science*, pages 273–287. Springer, 1997. doi:10.1007/3-540-63141-0_19.

33 Harold W. Kuhn. Extensive games and the problem of information. *Classics in Game Theory*, pages 46–68, 1953.

34 Orna Kupferman, Giuseppe Perelli, and Moshe Y. Vardi. Synthesis with rational environments. *Ann. Math. Artif. Intell.*, 78(1):3–20, 2016. doi:10.1007/S10472-016-9508-8.

35 Orna Kupferman and Noam Shenwald. The complexity of LTL rational synthesis. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*, volume 13243 of *Lecture Notes in Computer Science*, pages 25–45. Springer, 2022. doi:10.1007/978-3-030-99524-9_2.

36 Noémie Meunier. *Multi-Player Quantitative Games: Equilibria and Algorithms*. PhD thesis, Université de Mons, July 2016.

37 Robin Milner. An algebraic definition of simulation between programs. In D. C. Cooper, editor, *Proceedings of the 2nd International Joint Conference on Artificial Intelligence. London, UK, September 1-3, 1971*, pages 481–489. William Kaufmann, 1971. URL: http://ijcai.org/Proceedings/71/Papers/044.pdf.

38 Fabio Mogavero, Aniello Murano, Giuseppe Perelli, and Moshe Y. Vardi. What makes Atl* decidable? A decidable fragment of strategy logic. In Maciej Koutny and Irek Ulidowski, editors, *CONCUR 2012 - Concurrency Theory - 23rd International Conference, CONCUR 2012, Newcastle upon Tyne, UK, September 4-7, 2012. Proceedings*, volume 7454 of *Lecture Notes in Computer Science*, pages 193–208. Springer, 2012. doi:10.1007/978-3-642-32940-1_15.

39 Martin J. Osborne and Ariel Rubinstein. *A course in Game Theory*. MIT Press, Cambridge, MA, 1994.

40 Gary L. Peterson and John H. Reif. Multiple-person alternation. In *20th Annual Symposium on Foundations of Computer Science, San Juan, Puerto Rico, 29-31 October 1979*, pages 348–363. IEEE Computer Society, 1979. doi:10.1109/SFCS.1979.25.

41 Jean-François Raskin, Krishnendu Chatterjee, Laurent Doyen, and Thomas A. Henzinger. Algorithms for omega-regular games with imperfect information. *Log. Methods Comput. Sci.*, 3(3), 2007. doi:10.2168/LMCS-3(3:4)2007.

42 John H. Reif. The complexity of two-player games of incomplete information. *J. Comput. Syst. Sci.*, 29(2):274–301, 1984. doi:10.1016/0022-0000(84)90034-5.

43 Michael Ummels. Rational behaviour and strategy construction in infinite multiplayer games. In S. Arun-Kumar and Naveen Garg, editors, *FSTTCS 2006: Foundations of Software Technology and Theoretical Computer Science, 26th International Conference, Kolkata, India, December 13-15, 2006, Proceedings*, volume 4337 of *Lecture Notes in Computer Science*, pages 212–223. Springer, 2006. doi:10.1007/11944836_21.

44 Michael Ummels. The complexity of Nash equilibria in infinite multiplayer games. In Roberto M. Amadio, editor, *Foundations of Software Science and Computational Structures, 11th International Conference, FOSSACS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29 - April 6, 2008. Proceedings*, volume 4962 of *Lecture Notes in Computer Science*, pages 20–34. Springer, 2008. doi:10.1007/978-3-540-78499-9_3.

A Helpful Examples for Games on Graphs

In this appendix, we present additional examples and explanations for notions referred to in Section 2.

▶ **Example 22** (A simple reachability game)**.** Consider the game structure G pictured in Figure 3. Its initial state is v_0 and there are two players, player 0, who owns the circle states, and player 1, who owns the square state. Transitions are represented by the arrows

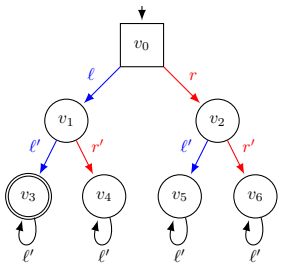


Figure 3 Two NEs and one SPE.

between states, such that there exists an arrow from a state v to another state v' if there exists an action a (either left or right here) such that $\delta(v, a) = v'$. The set of actions A is thus partitioned into $A_0 = \{\ell', r'\}$ and $A_1 = \{\ell, r\}$.

Both players have the same reachability objective, with $T_0 = T_1 = \{v_3\}$, denoted in the figure by the fact that state v_3 is double-circled. None of the two players have a winning strategy: player 0 can prevent player 1 to ever reach state v_3 by choosing to go to v_4, v_5 or v_6, depending on the first action of player 1, while player 1 can prevent player 0 from winning by going to state v_2.

▶ **Example 23** (A game with two NEs)**.** Let us come back to the reachability game of Figure 3. Two strategy profiles are represented in Figure 3: the first one, $\bar{\sigma}$, by the red transitions between states and the second one, $\bar{\sigma}'$, by the blue ones. Both profiles are NEs. Indeed, for the red profile with outcome $v_0 r v_2 r' (v_6 \ell')^\omega$, if player 0 deviates from σ_0, the resulting play $v_0 r v_2 \ell' (v_5 \ell')^\omega$ is still losing for him, since player 1 chooses to go to state v_2. On the other hand, if player 1 deviates from σ_1, the resulting play is $v_0 \ell v_1 r' (v_4 \ell')^\omega$, since player 0 chooses to go to state v_4, and this play is losing for player 1. Thus, none of the players have a profitable deviation from $\bar{\sigma}$. Notice that the gain profile of the red profile is equal to $(0, 0)$. Similarly, one can easily check that the blue profile is also an NE with a gain profile equal to $(1, 1)$.

In the red profile $\bar{\sigma}$ of the previous example, the possible choice of going from v_1 to v_4 for player 0 is irrational: after all, his objective is to reach state v_3. Notice, however, that this behavior is part of the NE $\bar{\sigma}$.

▶ **Example 24** (A game with an SPE)**.** Consider again the game in Figure 3. In the subgame $\mathcal{G}_{\restriction v_0 \ell v_1}$, starting after the history $v_0 \ell v_1$, the red profile is not an NE. Indeed, choosing to go to state v_4 is a non-credible threat from player 0, as his target set $\{v_3\}$ is accessible from state v_1, which he owns. The player 0 strategy that goes to v_3 from v_1 is thus a profitable deviation in the subgame $\mathcal{G}_{\restriction v_0 \ell v_1}$. Therefore, the red profile is not an SPE in $\mathcal{G}$. On the other hand, one can check that the blue profile is an SPE in $\mathcal{G}$, as it is an NE in every of its subgames.

▶ **Example 25** (A solution to the SPE-NCRS problem)**.** The answer to the SPE-NCRS problem for the game in Figure 3 is positive. Indeed, consider the strategy σ_0 of player 0 that chooses action ℓ' in both states v_1 and v_2. The unique subgame-perfect response $\bar{\sigma}_{-0}$ for player 1 is to choose action ℓ in v_0. The resulting strategy profile is thus the blue one whose outcome is winning for player 0.

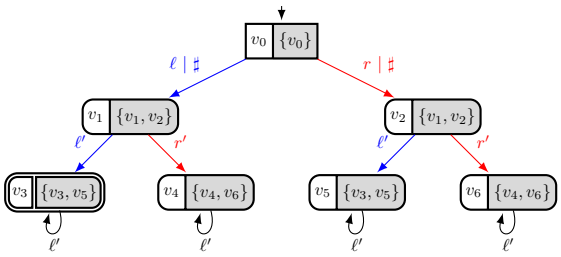


Figure 4 Imperfect information in a reachability game.

B SPE-NCRS Problem and P_1CP_2 Game

In this appendix, we provide additional material for Section 3.

B.1 Imperfect Information

▶ **Example 26** (Observation). In Figure 4, a two-player game structure is pictured, together with the observations of player 0, who owns the circle states (we suppose that player 1 has perfect information). Each state is divided in two sections: on the left, with a white background, the name of the state is displayed, while on the right side, with a gray background, the observation of player 0 is displayed. Similarly, the actions ℓ and r of player 1, who owns the square state, are accompanied by a $\#$ on their right side, which is an abuse of notation to mean $\{\ell, r\}$, that is, player 0 cannot distinguish between the two possible actions of player 1. Notice that from states owned by player 0, the actions do not have an observation attached to them, as player 0 knows his own actions, by hypothesis.

Let us now look at which histories player 0 can or cannot distinguish. From v_0, player 0 cannot distinguish between the actions of player 1, and both states v_1 and v_2 have the same observation set $\{v_1, v_2\}$. Therefore, the two histories $v_0\ell v_1$ and $v_0 r v_2$ are indistinguishable for player 0, i.e., $\mathcal{O}bs(v_0\ell v_1) = \mathcal{O}bs(v_0 r v_2)$. Similarly, we have $\mathcal{O}bs(v_0\ell v_1\ell' v_3) = \mathcal{O}bs(v_0 r v_2\ell' v_5)$.

▶ **Example 27** (Observation-based strategy). Let us come back to the game structure of Figure 4. Suppose that the objective of player 0 is to reach the target set $\{v_3\}$. One can verify that player 0 does not have an observation-based winning strategy. Indeed, as noted in Example 26, player 0 cannot distinguish between histories $h_1 = v_0\ell v_1$ and $h_2 = v_0 r v_2$. By definition, any observation-based strategy must prescribe the same action from both h_1 and h_2. If this action is r', then both resulting plays are losing for player 0. If it is ℓ', then one of the resulting play is winning, while the other is losing. Thus, no observation-based strategy ensures player 0 to reach $\{v_3\}$.

B.2 Details on the P_1CP_2 Game Structure

▶ **Definition 28** (Objectives of the P_1CP_2 game). *Let $\rho = v'_0 a'_0 v'_1 a'_1 v'_2 a'_2 \dots$ be a play in the game structure of $P_1CP_2(\mathcal{G})$. Let* $\mathsf{simGain}(\rho)$ *be its simulated gain.*

- *The play ρ is* winning for Challenger *if one of the following conditions is satisfied:*
  - (iC) *there exist $n \in \mathbb{N}$ and $g \in \{0,1\}^{|\Pi|}$, such that for every state v'_k with $k > n$, if v'_k is a player-state, then it is of the form $(v_k, i_k, \bar{g}_k)$ such that its player-component i_k equals $\varnothing$ and its gain-component satisfies*

$$\bar{g}_k = \mathsf{simGain}(\rho),$$

(***iiC***) *there exists a player* $i \neq 0$ *such that*
- *for every* $n \in \mathbb{N}$*, there exists* $k > n$ *for which* v'_k *is a player-state with its player-component being equal to* i*,*
- *there exists* $n \in \mathbb{N}$ *such that for every state* v'_k *with* $k > n$*, if* $v'_k = (v_k, i_k, \bar{g}_k)$ *is a player-state with* $i_k \neq \varnothing$*, then its player-component* i_k *equals* i *and the* i*-th component* $\bar{g}_{k,i}$ *of its gain-component* $\bar{g}_k$ *satisfies*

$$\bar{g}_{k,i} \geq \mathsf{simGain}(\rho)_i,$$

(***iiiC***) *there exist two distinct players* $i, j \neq 0$ *such that for every* $n \in \mathbb{N}$*, there exist* $k, \ell > n$ *for which* v'_k *and* v'_ℓ *are player-states, with their player-component being respectively* i *and* j*.*

The set of plays satisfying one of these conditions is denoted W_C*.*

- *The play* ρ *is* winning for Prover 1 and Prover 2 *if one of the following conditions is satisfied:*

(***iP***) *there exist* $n \in \mathbb{N}$ *and* $g \in \{0,1\}^{|\Pi|}$*, such that for every state* v'_k *with* $k > n$*, if* v'_k *is a player-state, then it is of the form* $(v_k, i_k, \bar{g}_k)$ *such that its player-component* i_k *equals* $\varnothing$ *and its gain-component satisfies*

$$\bar{g}_k \neq \mathsf{simGain}(\rho),$$

(***iiP***) *there exists a player* $i \neq 0$ *such that*
- *for every* $n \in \mathbb{N}$*, there exists* $k > n$ *for which* v'_k *is a player-state with its player-component being equal to* i*,*
- *there exists* $n \in \mathbb{N}$ *such that for every state* v'_k *with* $k > n$*, if* $v'_k = (v_k, i_k, \bar{g}_k)$ *is a player-state with* $i_k \neq \varnothing$*, then its player-component* i_k *equals* i *and the* i*-th component* $\bar{g}_{k,i}$ *of its gain-component* $\bar{g}_k$ *satisfies*

$$\bar{g}_{k,i} < \mathsf{simGain}(\rho)_i,$$

The set of plays satisfying one of these conditions is denoted W_P*.*

▶ Remark 29. We have $W_{P_1} = W_{P_2} = W_P$ and $W_P = \mathsf{Plays}_{G'} \setminus W_C$.

Languages of Boundedly-Ambiguous Vector Addition Systems with States

Wojciech Czerwiński
University of Warsaw, Poland

Łukasz Orlikowski
University of Warsaw, Poland

Abstract

The aim of this paper is to deliver broad understanding of a class of languages of boundedly-ambiguous VASSs, that is k-ambiguous VASSs for some natural k. These are languages of Vector Addition Systems with States with the acceptance condition defined by the set of accepting states such that each accepted word has at most k accepting runs. We develop tools for proving that a given language is not accepted by any k-ambiguous VASS. Using them we show a few negative results: lack of some closure properties of languages of k-ambiguous VASSs and undecidability of the k-ambiguity problem, namely the question whether a given VASS language is a language of some k-ambiguous VASS. In fact we show an even more general undecidability result stating that for any class containing all regular languages and only k-ambiguous VASS languages for some $k \in \mathbb{N}$ it is undecidable whether a language of a given 1-dimensional VASS belongs to this class. Finally, we show that the regularity problem is decidable for k-ambiguous VASSs.

2012 ACM Subject Classification Theory of computation → Parallel computing models

Keywords and phrases vector addition systems, Petri nets, unambiguity, bounded-ambiguity, languages

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2025.13

Related Version *Full Version*: https://arxiv.org/abs/2504.07669

Funding Both authors are supported by the ERC grant INFSYS, agreement no. 950398.

Acknowledgements We would like to thank David Purser for sharing with us a conjecture about language not recognised by an unambiguous VASS. We thank also Sławomir Lasota for suggesting us to reduce from 0-finite-reach instead of reducing from the halting problem for 2CM in the proof of Theorem 1. Additionally, we would like to thank Piotr Hofman for inspiring discussions and for reviewing the Master's Thesis [35], which served as the foundation for the results presented in this paper.

1 Introduction

Determinism is a central notion in computer science. Deterministic systems often allow for more efficient algorithms. On the other hand, usually deterministic systems are less expressive, so in many cases for a non-deterministic system there is no equivalent deterministic system and one cannot use more efficient techniques. For those reasons, there is recently a lot of research devoted to various notions restricting nondeterminism in a milder way than determinism. The hope is that systems having the considered properties are more expressive than the deterministic ones, but still allow for robust algorithms design. One prominent example of such a notion is unambiguity; a system is unambiguous if for every word there is at most one accepting run over this word. In the last decade unambiguous systems were intensely studied and for various classes of infinite-state systems the unambiguous case turns out to be much more tractable [8, 9, 18, 25, 29]. Similar notions were also investigated recently,

© Wojciech Czerwiński and Łukasz Orlikowski;
licensed under Creative Commons License CC-BY 4.0

like k-ambiguity (each word is accepted by at most k runs) and history-determinism (a weakened version of determinism), in both cases one can design more efficient algorithms in some cases [7, 10].

In this paper we focus on studying milder version of determinism for Vector Addition Systems with States (VASS). VASSs and related Petri nets are popular and fundamental models of concurrency with many applications both in theory and in practical modelling [30, Section 5]. Languages of VASSs with restricted nondeterminism were already studied for several years, mostly with the acceptance condition being the set of accepting states. In [9] it was shown that the universality problem for unambiguous VASSs is decidable in ExpSpace, in contrast to Ackermann-completeness of the problem for VASSs without that restriction. In [10, 11] the language equivalence problem was considered for unambiguous VASS and more generally for k-ambiguous VASSs and it was shown to be decidable and Ackermann-complete, in contrast to undecidability in general [2], even in dimension one [21, Thm. 20]. The choice of universality and equivalence problems is deliberate here. The complexity of the seemingly more natural language emptiness problem (or equivalently of the reachability problem) does not depend on the unambiguity assumption. Indeed, one can always transform a system to a deterministic one by assigning all the transitions unique labels without affecting nonemptiness. Therefore, in order to observe a difference in complexity or decidability after restricting nondeterminism, one should consider problems, which do not simply ask about the existence of some object. Some natural problems concerning languages of that type are the universality problem and the language equivalence problem and these two problems were extensively studied for many models with the unambiguity assumption or some modification of it. First of all, the universality and equivalence problems are solvable in PTime [32] (and even in NC^2 [33]) for unambiguous finite automata (UFA), in contrast to PSpace-hardness of both problems for NFA. The universality problem was also shown decidable for unambiguous register automata in a sequence of papers [3, 14, 25] with improving complexity, which contrasts undecidability in the case without restricted nondeterminism [26, Thm 5.1]. This line of research culminated in a very elegant contribution [5, 6], which showed in particular that not only the universality problem, but also the equivalence problem is solvable in ExpTime for unambiguous register automata, and in PTime in the case of fixed number of registers. Another popular restriction of nondeterminism was also studied recently: history-determinism. A system is history-deterministic if its nondeterminism can be resolved on the fly, based on the history of a particular run. The equivalence and inclusion problems were shown to be decidable for one-dimensional history-deterministic VASSs [27], but undecidable for two-dimensional history-deterministic VASSs [7].

Despite a lot of research on algorithms for unambiguous and boundedly-ambiguous (k-ambiguous for some $k \in \mathbb{N}$) VASS not much is known about the class of languages recognisable by such models. In particular, to our best knowledge, till now it was even not known whether there exist any VASS language (we consider acceptance by states), which is not a language of an unambiguous VASS. The reason behind this lack of knowledge was absence of any technique, which can show that a given language of a VASS cannot be recognised by an unambiguous VASS. The quest for such a technique is natural and the question deserves investigation. Analogous problems were considered for other models of computation. For finite automata the question trivialises, as deterministic automata recognise all the regular languages. However, already for weighted automata over a field the problems are highly nontrivial and were recently studied in-depth [4, 28], in particular it is decidable whether a given weighted automaton is unambiguisable, so equivalent to some unambiguous weighted automaton [4]. The problem whether a given context-free language is unambiguisable is

known to be undecidable since the 60s [17, 19]. The aim of this paper is deepening the understanding of the class of languages recognisable by boundedly-ambiguous VASS and its subclasses.

Our contribution. In the paper we deliver several results, which help understanding unambiguous, k-ambiguous and boundedly-ambiguous VASS languages. Our first main tool is Lemma 5, which delivers the first example of a VASS language, which is known to be not a k-ambiguous VASS language. The second main tool is Lemma 8, which formulates a condition, which needs to be satisfied by all k-ambiguous VASS languages. Using these two lemmas we can rather straightforwardly inspect closure properties of k-ambiguous VASS languages in Lemma 10 and show several expressivity results in Section 4.2. Further building on Lemma 5 we obtain our main contribution.

▶ **Theorem 1.** *For any class $\mathcal{C}$ of languages containing all regular languages and contained in the class of all boundedly-ambiguous VASS languages it is undecidable to check whether the language of a given 1-VASS accepting by states belongs to $\mathcal{C}$.*

Consequences of Theorem 1 are broad. It reproves undecidability of regularity of 1-dimensional VASSs (in short 1-VASSs) [12, Section 8] and undecidability of determinisability of 1-VASSs considered in [1]. It is important to emphasise that Theorem 1 gives us extensive flexibility wrt. the undecidability results. For example in [1] it was shown that for a given 1-VASS it is undecidable whether there is an equivalent deterministic 1-VASS. One can argue that possibly asking about equivalent deterministic VASS, without a bounding dimension, is a more natural question, which deserves independent research. Theorem 1 answers negatively all questions of that kind in one shot. We formulate below its corollary to illustrate variety of consequences we obtain. In particular we know now that for many classical restrictions of nondeterminism it is undecidable whether for a given 1-VASS there exists some equivalent one with nondeterminism restricted in that way.

▶ **Corollary 2.** *It is undecidable whether the language of a given 1-VASS accepting by states is recognisable by some*

- *unambiguous VASS,*
- *k-ambiguous VASS for given $k \in \mathbb{N}$,*
- *boundedly ambiguous VASS,*
- *deterministic VASS,*
- *k-ambiguous 1-VASS for given $k \in \mathbb{N}$.*

Our last main contribution is Theorem 22, which states that the regularity problem is decidable for boundedly-ambiguous VASSs, which contrasts undecidability without that assumption [2,21]. One can see this result as an intuitive indication that boundedly-ambiguous VASSs are closer to deterministic VASSs rather than to general nondeterministic VASSs.

Organisation of the paper. In Section 2 we introduce preliminary notions and recall useful lemmas. Section 3 is devoted to showing the two main technical tools, namely Lemmas 5 and 8. In Section 4 we present closure properties and expressivity results for the classes of k-ambiguous VASSs. Next, in Section 5 we show our main result, namely Theorem 1. Theorem 22 is proved in Section 6. Finally, in Section 7 we discuss interesting future research directions.

2 Preliminaries

Basic notions. For $a, b \in \mathbb{N}$ such that $a \leq b$ we write $[a, b]$ for the set of integers $\{a, a+1, \ldots, b\}$. For $a \in \mathbb{N}$ we write $[a]$ for the set $[1, a]$. For a vector $v \in \mathbb{Z}^d$ we write v_i for the i-th entry of v. For a vector $v \in \mathbb{Z}^d$ we write $\text{support}(v)$ for the set $\{i \mid v_i > 0\}$. For two vectors $u, v \in \mathbb{N}^d$ we write $u \leq v$ if for all $i \in [d]$ we have $u_i \leq v_i$. We also extend this order to $(\mathbb{N} \cup \{\omega\})^d$ where ω is bigger than any natural number. We write $\mathbb{N}_\omega$ for the set $\mathbb{N} \cup \{\omega\}$. Whenever we speak about the norm of vector $v \in \mathbb{Z}^d$ we mean $||v|| = \max_{1 \leq i \leq d} |v_i|$. For a word w we denote by $\#_a(w)$ the number of letters a in the word w.

Downward-closed sets. A set $S \subseteq N^d$ is downward-closed if for each $u, v \in \mathbb{N}^d$ it holds that $u \in S$ and $v \leq u$ implies $v \in S$. For $u \in \mathbb{N}^d$ we write $u\downarrow$ for the set $\{v \mid v \leq u\}$. If a downward-closed set is of the form $u\downarrow$ we call it a down-atom. Observe, that a one-dimensional set $S \subseteq \mathbb{N}$ is downward-closed if either $S = \mathbb{N}$ or $S = [0, n]$ for some $n \in \mathbb{N}$. Thus we have either $S = \omega\downarrow$ in the first case or $S = n\downarrow$ in the second case. So equivalently a downward-closed set $D \subseteq \mathbb{N}^d$ is a down-atom if it is of a form $D = D_1 \times \ldots \times D_d$, where for all $i \in [d]$ we have that D_i is a downward-closed one dimensional set. For simplicity we write $D = (n_1, n_2, \ldots, n_d)\downarrow$ if $D = n_1\downarrow \times n_2\downarrow \ldots n_d\downarrow$. Therefore each down-atom is of the form $u\downarrow$ where $u \in (\mathbb{N} \cup \omega)^d$. The following proposition will be helpful in our considerations.

▶ **Proposition 3** (Lemma 17 in [13], [16]). *Each downward-closed set in $\mathbb{N}^d$ is a finite union of down-atoms.*

Vector Addition Systems with States. A d-dimensional Vector Addition System with States (d-VASS) is a nondeterministic finite automaton with d non-negative integer counters. Transitions of the VASS manipulate these counters. Formally, we define VASS V as $V = (\Sigma, Q, \delta, I, F)$ where Σ is a finite alphabet, Q is a finite set of automaton states, $\delta \subseteq Q \times (\Sigma \cup \varepsilon) \times \mathbb{Z}^d \times Q$ is a transition relation, $I \subseteq Q \times \mathbb{N}^d$ is a finite set of initial configurations and $F \subseteq Q \times \mathbb{N}^d$ is a finite set of final configurations. For a transition $t = (s, a, v, s') \in \delta$ we say that the transition is over a or the transition reads letter a. We also write $\text{EFF}(t)$ for the effect of transition, which is v. We define the norm of transition t as the norm of v.

A d-VASS can be seen as an infinite-state labelled transition system in which each configuration is a pair $(s, u) \in Q \times \mathbb{N}^d$. We denote such configuration as $s(u)$. We define the norm of the configuration as the norm of u. A transition $t = (s, a, v, s') \in \delta$ can be fired in a configuration $q(u)$ if and only if $q = s$ and $u' \geq 0$ where $u' = u + v$. After firing the transition configuration is changed to $s'(u')$. We also define run as a sequence of transitions, which can be fired one after another from some configuration. For a run $\rho = t_1 t_2 \ldots t_n$ we write $\text{EFF}(\rho)$ for $\Sigma_{i=1}^{n} \text{EFF}(t_i)$. If we want to say something only about jth (for $j \in [d]$) entry of the $\text{EFF}(\rho)$ we write $\text{EFF}_j(\rho)$. We also define $\text{support}(\rho)$ as $\text{support}(\text{EFF}(\rho))$. We say that a run ρ is from configuration $q(u)$ to $p(u')$ if the sequence of transitions can be fired from $q(u)$ and the final configuration is $p(u')$. We say that a run is a loop if the state of its initial configuration is the same as the state of the final configuration. For two runs $\rho_1 = \alpha_1 \ldots \alpha_n$ and $\rho_2 = \beta_1 \ldots \beta_k$ if the sequence $\alpha_1 \ldots \alpha_n \beta_1 \ldots \beta_k$ is a run ρ then we can write $\rho = \rho_1 \rho_2$. We say, that a run $\rho = t_1 \ldots t_n$ is over $w = \lambda_1 \ldots \lambda_n \in (\Sigma \cup \{\varepsilon\})^*$ (or reads w) if and only if for each $i \in [n]$ transition t_i is over λ_i. We denote by $w(\rho)$ the word read by ρ. We say that the length of a run ρ is equal to n if it consists of n transitions. For the length of a run, we write $|\rho|$.

We say, that VASS is ε-free if there is no transition over ε. In this work, unless stated otherwise, we work with ε-free VASSs.

Languages of VASSs. A run of a VASS is accepting if it starts in an initial configuration $c_0 \in I$ and ends in an accepting configuration. A configuration is accepting if it covers some configuration from the set of final configurations F. In other words configuration $q(v)$ is accepting if there exists a configuration $q(v') \in F$ such that $v \geq v'$. For a VASS V we define its language as the set of all words read by accepting runs and we denote it by $L(V)$. Languages defined in this way are called coverability languages. Sometimes we consider languages with other acceptance condition (reachability languages), in that case we indicate it clearly.

We sometimes consider reachability languages in which run ending in configuration $q(v)$ is accepting if and only if $q(v) \in F$. Sometimes, we consider VASSs, where the set of accepting configurations is infinite, but has a specific form. For instance we consider downward-VASSs, where the set of accepting configurations is possibly infinite, but it is downward-closed. In this type of VASSs a run ending in configuration $q(v)$ if and only if $q(v) \in F$ where F is a downward-closed set of accepting configurations. We say, that VASS V is deterministic if it has only one initial configuration, it is ε-free and for each state q and each letter $a \in \Sigma$ there is at most one transition over a leaving the state q. We say, that VASS V is k-ambiguous if and only if for every $w \in L(V)$ we have at most k accepting runs over w. We also say then, that $L(V)$ is a k-ambiguous language. For a d-VASS consider a function $r : (Q \times \mathbb{N}^d \times \delta)^* \times (Q \times \mathbb{N}^d) \times \Sigma \to \delta^*$ that, given a history of the run (configurations and taken transitions), current configuration $q(v)$ and a next letter $\lambda \in \Sigma$, returns a sequence of possibly many transitions. One of these transitions is over λ, all the other are over ε, and the sequence can be fired from the current configuration $q(v)$. Let us call r a resolver. We say, that d-VASS V is history-deterministic if and only if it has one initial configuration and there exists a resolver r such that for each $w \in L(V)$ run ρ over w from the initial configuration given by the resolver is accepting.

We denote by Det, Hist, k-Amb, BAmb and NonDet the class of languages of respectively deterministic, history-deterministic, k-ambiguous , all boundedly-ambiguous and fully non-deterministic VASSs languages. We sometimes denote by 1-Amb the class of unambiguous VASSs languages. By d-NonDet we denote the class of languages of d-VASSs.

3 Tools for separating BAmb and NonDet

In this section we develop two techniques for showing that a language is not recognised by a k-ambiguous VASS: Lemma 5 and Lemma 8.

3.1 Dominating block

The first tool is based on the following observation about a language not recognised by a k-ambiguous VASS.

▶ **Lemma 4.** *For every $k \in \mathbb{N}_+$ the language*

$$L_k = \{a^{n_1} b a^{n_2} b a^{n_3} b \dots a^{n_{k+2}} \mid \exists_{i \in [k+1]}\ n_i \geq n_{i+1}\}$$

is not recognised by a k-ambiguous VASS.

For showing various properties of boundedly-ambiguous VASS Lemma 4 is sufficient. In one case, for proving Theorem 1, we need a strengthening of Lemma 4, which is presented in the following lemma:

▶ **Lemma 5.** *Let Σ be an alphabet such that $b \notin \Sigma$ and let L be a language over Σ. For each function $f : L \to \mathbb{N}_\omega$ such that $\sup f = \omega$ (recall that $\sup f = \sup\{f(x) \mid x \in L\}$) language $L_1 = \{a^{n_1}ba^{n_2}ba^{n_3}b \dots a^{n_{k+2}}bw \mid w \in L, \exists_{i\in[k+1]}\ n_i \geq n_{i+1} \vee n_{k+2} \geq f(w)\}$ is not recognised by a k-ambiguous VASS.*

Before proving Lemma 5 we show how it implies Lemma 4.

Proof of Lemma 4. Let us fix $k \in \mathbb{N}_+$ and let $L = \{\varepsilon\}$. Let $f : L \to \mathbb{N}_\omega$ be defined as $f(\varepsilon) = \omega$. Hence by Lemma 5 we get that L_k is not recognised by a k-ambiguous VASS. ◀

Below we sketch the idea behind the proof of Lemma 5. We assume, for the sake of contradiction, that L_1 is recognised by a k-ambiguous VASS and aim at a contradiction by showing $k+1$ different runs over the same word. We first consider $k+1$ words $w_1, \dots, w_{k+1} \in L_1$, where $u \in L$ is a particularly chosen word and $N_0 < N_1 < \dots < N_{k+2} \in \mathbb{N}$ are particularly chosen constants:

$$w_i = a^{N_1!}ba^{N_2!}b \dots a^{N_i!}ba^{N_0!}ba^{N_{i+2}!}ba^{N_{i+3}!}b \dots a^{N_{k+2}!}bu.$$

Then we dive into combinatorics of VASS runs ρ_i over words w_i and conclude that there are specific pumping cycles in ρ_i. We show two main lemmas, which analyse the structure of the runs. The first one gives conditions for finding a specific loop in a run. Observe, that this lemma is general and is not restricted to k-ambiguous VASSs.

▶ **Lemma 6.** *For each d-dimensional VASS V, each subset of counters $S \subseteq [d]$ and each $n \in \mathbb{N}$ there exists a constant $M := M(V, S, n) \geq 1$ such that in every run in V, starting from a configuration in which values of counters from S are at most n, which is of length at least M there exists a loop, which has non-negative effect on the counters from S.*

The proof of Lemma 6 uses rather standard techniques similar to the ones in the Karp-Miller construction solving the coverability problem for VASS [22]. Lemma 6 is used to find loops in words w_i. For example one can show that if N_1 is big enough then one can find a loop as in Lemma 6 in the first block of letters a. Similarly, if N_2 is big enough wrt. to N_1, one can find an appropriate loop in the second block of letters a. Using this approach one can find loops in blocks of letters a, if the block is much longer than the whole prefix before it. This is however not true for the block of length N_0. Therefore we need a more sophisticated tool in order to be able to modify runs over all the words w_i to runs over the same word.

This requires the more challenging part of the proof, which provides a detailed characterisation of runs of k-ambiguous VASS. We formulate below one of the used lemmas in order to illustrate the kind of arguments we consider and, as it is also used in Section 3.2.

▶ **Lemma 7.** *Let L be a language over $\Sigma = \{a, b\}$ recognised by some k-ambiguous d-VASS V. For each $n \in \mathbb{N}$ there exists a constant C such that each run ρ in V such that:*

- *Run ρ is a prefix of an accepting run.*
- *Run ρ is reading $a^{m_1}ba^{m_2}b \dots a^{m_{n-1}}ba^{m_n}$.*

can be decomposed as:

1. *$\rho = \alpha_1\beta_1^{a_1}\alpha_1'\alpha_2\beta_2^{a_2}\alpha_2' \dots \alpha_n\beta_n^{a_n}\alpha_n'$ for some $a_1, a_2, \dots, a_n \geq 1$ and for each $i \in [1, n]$ we have $|\alpha_i|, |\beta_i|, |\alpha_i'| \leq C$.*
2. *For each $j < n$ we have $w(\alpha_j') \in L(a^*b)$ and $w(\alpha_n') \in L(a^*)$*
3. *For each $j \in [n]$ we have that $w(\alpha_j), w(\beta_j) \in L(a^*)$*
4. *For each $j \in [n]$ we have that β_j is either a loop or ε. Moreover if $m_j \geq 2 \cdot C + 1$ then $\beta_j \neq \varepsilon$.*

5. *For each* $j \in [n]$ *let* $A_j = \bigcup_{1 \leq i < j} \text{support}(\beta_i)$ *then* β_j *is nonnegative on counters from* $[d] \setminus A_j$
6. *For each* $j \in [n]$ *there is no* λ, δ *and* λ' *such that* δ *is a nonnegative loop on counters from* $[d] \setminus A_j$, $\lambda' \neq \varepsilon$ *and* $\lambda\delta\lambda' = \alpha_j\beta_j$

The proof of Lemma 7 uses the Lemma 6 and a lot of combinatorial observations. It is moved to the Appendix.

By appropriate use of Lemma 7 and other auxiliary lemmas we show that ρ_i can be modified a bit into runs ρ'_i, which are all different, all accepting and all over the same word w, where

$$w = a^{n_1}ba^{n_2}b \dots a^{n_{k+2}}bu$$

and $n_j = 2m^{k+3-j}\Pi_{l=j}^{k+2} N_l!$ (where m is the maximal norm of a transition). More concretely, ρ'_i is obtained by pumping the loops β_i in ρ_i more times. A challenge it to show that the resulting ρ'_i are indeed different, which we achieve by more careful, but technical investigation of the runs. Existence of $k+1$ different runs over the same word w is a contradiction with the assumption that L_1 is recognised by a k-ambiguous VASS and finishes the proof.

3.2 Semilinear image

Now, we develop the second tool for showing that language is not recognised by a k-ambiguous VASS. Before formulating the tool we have to provide a few definitions. For any language $L \subseteq \{a, b\}^*$ such that for each $w \in L$ we have $\#_b(w) = l$ for some $l \in \mathbb{N}$ we define

$$im(L) = \{(a_1, a_2, \dots, a_{l+1}) \mid a_1, a_2, \dots, a_{l+1} \in \mathbb{N}, a^{a_1}ba^{a_2}b \dots ba^{a_{l+1}} \in L\}$$

Given a base vector $b \in \mathbb{Z}^d$ and a finite set of period vectors $P = \{p_1, \dots, p_n\} \subseteq \mathbb{Z}^d$, the linear set $L(b, P)$ is defined as

$$L(b, P) = \{b + a_1p_1 + \dots + a_np_n \mid a_i \in \mathbb{N}, 1 \leq i \leq n\}$$

A semi-linear set is a finite union of linear sets. Now we are ready to formulate the second tool.

▶ **Lemma 8.** *Let* $L \subseteq \{a, b\}^*$ *be a language satisfying:*
- *L is recognised by k-ambiguous VASS V.*
- *There exists $n \in \mathbb{N}$ such that for each $w \in L$ we have $\#_b(w) = n$.*

Then $im(L)$ is a semilinear set.

Proof. The proof is based on Lemma 7 and the fact that set of solution of a system of linear Diophantine inequalities is semilinear. Notice first, that L satisfies conditions of Lemma 7. Therefore we can apply Lemma 7 to L and $n+1$ and decompose each accepting run in V in the way presented in Lemma 7. Let us fix some $\alpha_1, \dots, \alpha_{n+1}$, $\alpha'_1, \dots, \alpha'_{n+1}$ and $\beta_1, \dots, \beta_{n+1}$, initial configuration $q(c)$ and accepting configuration $p(f)$. We call a run ρ accepting with respect to $q(c)$ and $p(f)$ if ρ is an accepting run of the VASS starting in $q(c)$ and ending in configuration $p(c')$ such that $c' \geq f$. Let K be the following language:

$$K = \{w \in L \mid \text{there exist } a_1, a_2, \dots, a_{n+1} \in \mathbb{N} \text{ such that } \alpha_1\beta_1^{a_1}\alpha'_1\alpha_2\beta_2^{a_2}\alpha'_2 \dots \alpha_{n+1}\beta_{n+1}^{a_{n+1}}\alpha'_{n+1}$$

$$\text{is an accepting run with respect to } q(c) \text{ and } p(f) \text{ and reads } w\}$$

Observe, that K depends on chosen α_i, β_i and α'_i. Moreover, observe, that because of the constant given by Lemma 7 we have only a finite number of possibilities of $\alpha_1, \dots, \alpha_{k+1}$,

$\alpha'_1, \ldots, \alpha'_{k+1}$, $\beta_1, \ldots, \beta_{k+1}$, initial configuration $q(c)$ and accepting configuration $p(f)$. Moreover, semilinear sets are closed under a finite union. Therefore to conclude, that $im(L)$ is a semilinear set it is enough to show that $im(K)$ is a semilinear set. Notice, that from conditions of Lemma 7, we know, that only α'_i (for $i \in [n]$) contain letter b, each exactly one letter at the last position. Hence:

$$im(K) = \{(|\beta_1|a_1 + |\alpha_1| + |\alpha'_1| - 1, |\beta_2|a_2 + |\alpha_2| + |\alpha'_2| - 1, \ldots,$$

$$, |\beta_n|a_n + |\alpha_n| + |\alpha'_n| - 1, |\beta_{k+1}|a_{n+1} + |\alpha_{n+1}| + |\alpha'_{n+1}|) \mid$$

such that $\alpha_1\beta_1^{a_1}\alpha'_1\alpha_2\beta_2^{a_2}\alpha'_2 \ldots \alpha_{n+1}\beta_{n+1}^{a_{k+1}}\alpha'_{n+1}$ is an accepting run with respect to $q(c)$ and $p(f)\}$

Therefore it is enough to show, that:

$$A = \{(a_1, a_2, \ldots, a_{n+1}) \mid \text{such that}$$

$$\alpha_1\beta_1^{a_1}\alpha'_1\alpha_2\beta_2^{a_2}\alpha'_2 \ldots \alpha_{n+1}\beta_{n+1}^{a_{n+1}}\alpha'_{n+1}$$

is an accepting run with respect to $q(c)$ and $p(f)\}$

is a semilinear set. We have two cases. Either $A = \emptyset$, hence semilinear. This case occurs if for any $a_1, a_2, \ldots, a_{n+1} \geq 1$ we do not have an accepting run with respect to $q(c)$ and $p(f)$. Otherwise, we will show semilinearity, by providing a system of linear inequalities for $a_1, a_2, \ldots, a_{n+1}$. It is enough because in [20] it was shown, that the set of solutions of a system of linear inequalities is a semilinear set. The goal of this system of linear inequalities is to express, that after each prefix of a run, we are non-negative on all of the counters. Moreover, we want also to express the acceptance condition. Therefore for each counter i we write the following inequalities:

- For each transition t and each α_j such that there exist u and v such that $\alpha_j = utv$:

$$c_i + \textsc{eff}_i(\alpha_1\beta_1^{a_1}\alpha'_1\alpha_2\beta_2^{a_2}\alpha'_2 \ldots \alpha_{j-1}\beta_{j-1}^{a_{j-1}}\alpha'_{j-1}) + \textsc{eff}_i(ut) \geq 0$$

- For each transition t and each α'_j such that there exist u and v such that $\alpha'_j = utv$:

$$c_i + \textsc{eff}_i(\alpha_1\beta_1^{a_1}\alpha'_1\alpha_2\beta_2^{a_2}\alpha'_2 \ldots \alpha_{j-1}\beta_{j-1}^{a_{j-1}}\alpha'_{j-1}\alpha_j\beta_j^{a_j}) + \textsc{eff}_i(ut) \geq 0$$

- For each transition t and each β_j such that there exist u and v such that $\beta_j = utv$:
  - If $\textsc{eff}_i(\beta_j) \leq 0$:

$$c_i + \textsc{eff}_i(\alpha_1\beta_1^{a_1}\alpha'_1\alpha_2\beta_2^{a_2}\alpha'_2 \ldots \alpha_{j-1}\beta_{j-1}^{a_{j-1}}\alpha'_{j-1}\alpha_j\beta_j^{a_j-1}) + \textsc{eff}_i(ut) \geq 0$$

  - Otherwise:

$$c_i + \textsc{eff}_i(\alpha_1\beta_1^{a_1}\alpha'_1\alpha_2\beta_2^{a_2}\alpha'_2 \ldots \alpha_{j-1}\beta_{j-1}^{a_{j-1}}\alpha'_{j-1}\alpha_j) + \textsc{eff}_i(ut) \geq 0$$

- Acceptance condition:

$$c_i + \textsc{eff}_i(\alpha_1\beta_1^{a_1}\alpha'_1\alpha_2\beta_2^{a_2}\alpha'_2 \ldots \alpha_{k+1}\beta_{n+1}^{a_{n+1}}\alpha'_{n+1}) \geq f_i$$

- Condition, that each a_i is positive (this is needed because of conditions of Lemma 7): $a_i \geq 1$

In other words, this system of inequalities ensures, that each transition in the sequence can be fired, check the acceptance condition and ensures that each a_i is positive. We have shown, that the set A is semilinear and hence $im(K)$ is a semilinear set. Therefore, because semilinar sets are closed under a finite union we have that $im(L)$ is a semilinear set. ◀

Then, as shown in Lemma 10, k-ambiguous VASS languages are closed under intersection with regular languages. As mentioned below languages K_n are regular for each $n \in \mathbb{N}$, the following corollary holds:

▶ **Corollary 9.** *Let $a, b \in \Sigma$, $L \subseteq \Sigma^*$ be a language recognised by a k-ambiguous VASS and for $n \in \mathbb{N}$ let $K_n \subseteq \{a, b\}^*$ be the language of words containing exactly n letters b. Then for any $n \in N$ we have that $im(L \cap K_n)$ is a semilinear set.*

4 Properties

In this section we present several properties of languages of boundedly-ambiguous Vector Addition Systems with States.

4.1 Closure properties

First, we investigate the closure properties of boundedly-ambiguous languages.

▶ **Lemma 10.** *If L_1 and L_2 are recognised by a k_1-ambiguous and k_2-ambiguous VASS respectively then:*

- *$L_1 \cap L_2$ is recognised by a $(k_1 \cdot k_2)$-ambiguous VASS;*
- *$L_1 \cup L_2$ is recognised by a $(k_1 + k_2)$-ambiguous VASS.*

Moreover, class of languages of boundedly-ambiguous VASSs is not closed under complementation and commutative closure.

Proof. We split the proof into several parts, each corresponding to the closure properties under one operation.

Intersection. Let L_1 and L_2 be languages recognised respectively by d_1-dimensional k_1-ambiguous VASS A_1 and d_2-dimensional k_2-ambiguous VASS A_2. Language $L_1 \cap L_2$ can be recognised by the standard synchronised product of A_1 and A_2. It is easy to observe that the product is a $(k_1 \cdot k_2)$-ambiguous $(d_1 + d_2)$-VASS.

Union. Let L_1 and L_2 be languages recognised by d_1-dimensional k_1-ambiguous VASS A_1 and d_2-dimensional k_2-ambiguous VASS A_2. The idea is to recognise $L_1 \cup L_2$ by taking union VASS $A_1 \cup A_2$, which is clearly $k_1 + k_2$-ambiguous and $\max(d_1, d_2)$-dimensional.

Complementation. This comes from a general fact, that coverability languages are not closed under complementation. Language $L = a^n b^{\leq n}$ is recognised even by a deterministic VASS (hence also by one with bounded-ambiguity). On the other hand, in [13] it was shown, that every two disjoint coverability VASS[1] languages L_1 and L_2 are regular separable, which means there exists a regular language L_3 such that $L_1 \subseteq L_3$ and $L_2 \cap L_3 = \emptyset$. Because L is a coverability VASS language its complement can be a coverability VASS language if and only if L is a regular language. Clearly L_1 is not and this can be shown using the Pumping Lemma for regular languages.

Commutative closure. Boundedly-ambiguous languages are not closed under commutative closure. Let us consider language $L = a^n b^{\leq n}$, which is recognised by a deterministic VASS. Its commutative closure is equal to $L_1 = \{w \mid \#_a(w) \geq \#_b(w)\}$. Assume, towards contradiction, that L_1 is recognised by a k-ambiguous VASS for $k \in \mathbb{N}_{\geq 0}$. Because boundedly-ambiguous languages are closed under intersection with regular languages also language

[1] In fact this result was shown for a wider class of Well-structured transition system (WSTS)

$L_1 \cap L(b^*a^*) = b^n a^{\geq n}$ is recognised by a k-ambiguous VASS. Let V be k-ambiguous VASS recognising $b^n a^{\geq n}$. By Lemma 6 we can take N such, that while accepting $b^N a^N$ VASS V will fire a non-negative loop on all of the counters. This loop reads b^l for some $l \in \mathbb{N}$. Because VASS V is k-ambiguous $l \geq 1$. Hence we can fire this loop one more time and accept $b^{N+l}a^N$, which is not in the language $b^n a^{\geq n}$. Therefore we reached a contradiction and boundedly-ambiguous languages are not closed under commutative closure. ◀

Using the fact that regular languages are unambiguous (i.e. 1-ambiguous) VASS languages and Lemma 10 we can formulate the following remark about closure of boundedly-ambiguous languages under intersection with unambiguous (hence also regular) languages.

▶ Remark 11. For each $k \in \mathbb{N}_+$ the class of k-ambiguous VASS languages is closed under intersection with unambiguous VASS languages. Hence, the same holds for the intersection with regular languages.

We complement Lemma 10 with Lemma 12 and Conjecture 13.

▶ **Lemma 12.** *For each $k_1, k_2 \in \mathbb{N}_+$ there exist languages L_1, L_2, which are respectively recognised by a k_1 and k_2 ambiguous VASS such that language $L_1 \cup L_2$ is not recognised by an n-ambiguous VASS for $n \in [k_1 + k_2 - 1]$.*

Proof. Let $k = k_1 + k_2$. For $i \in [k]$ let us define language $U_i = \{a^{n_1}ba^{n_2}ba^{n_3}b \dots a^{n_{k+1}} \mid n_i \geq n_{i+1}\}$. Observe, that U_i is a language of an unambiguous VASS. Let $L_1 = \bigcup_{i=1}^{k_1} U_i$ and $L_2 = \bigcup_{i=k_1+1}^{k}$. By Lemma 10 languages L_1 and L_2 are respectively recognised by a k_1 and k_2 ambiguous VASS. By applying Lemma 4 to $k-1$ we get that language $L_1 \cup L_2$ is not recognised by a n-ambiguous VASS for $n \in [k-1]$. ◀

▶ **Conjecture 13.** *For each $k_1, k_2 \in \mathbb{N}_+$ there exists languages L_1, L_2, which are respectively recognised by a k_1 and k_2 ambiguous VASS such that language $L_1 \cap L_2$ is not recognised by a n-ambiguous VASS for $n \in [k_1 \cdot k_2 - 1]$.*

4.2 Expressiveness

Firstly, we show that the hierarchy of k-ambiguous VASS is strict.

▶ **Lemma 14.** *For every $k \in \mathbb{N}_+$ there exists language $L \in$ (k+1)-Amb $\setminus$ k-Amb.*

Proof. Let $L = \{a^{n_1}ba^{n_2}ba^{n_3}b \dots a^{n_{k+2}} \mid \exists_{i \in [k+1]}\ n_i \geq n_{i+1}\}$. Because of Lemma 4 language L is not recognised by a k-ambiguous VASS. On the other hand it is a union of $k+1$ unambiguous VASS languages. For $i \in [k+1]$ let us define $L_i = \{a^{n_1}ba^{n_2}ba^{n_3}b \dots a^{n_{k+2}} \mid n_i \geq n_{i+1}\}$. Observe, that $L = \bigcup_{i=1}^{k+1} L_i$. Hence, by Lemma 10, L is recognised by a $(k+1)$-ambiguous VASS. ◀

In terms of the classes of languages they define, boundedly-ambiguous VASS can express strictly more than deterministic ones. Moreover, they are strictly less expressive than non-deterministic ones. We prove this in Lemmas 15 and 17.

▶ **Lemma 15.** *There exists language $L \in$ 1-Amb $\setminus$ Det.*

We show an even stronger Lemma 16. It is stronger because we have Det $\subseteq$ Hist.

▶ **Lemma 16.** *There exists language $L \in$ 1-Amb $\setminus$ Hist.*

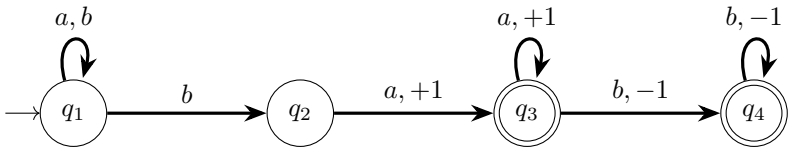


Figure 1 Unambiguous VASS recognising (starting from zero) $\{a,b\}^*ba^{n>0}b^{\leq n}$.

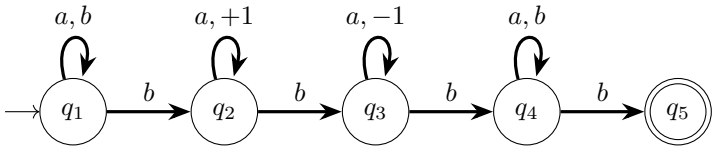


Figure 2 VASS recognising (starting from zero) $\{a^nba^mba^k \mid n,m,k \in \mathbb{N}, (n \geq m \vee n \geq k)\}$.

Proof. Let $L = \{a,b\}^*ba^{n>0}b^{\leq n}$. Observe, that L is recognized by an unambiguous VASS depicted in Figure 1. The only point of nondeterminism is in state q_1 and there is only one way to guess when the last block, ending the word in the form $a^{n>0}b^{\leq n}$ comes.

On the other hand, assume, towards contradiction, that L is recognised by a history-deterministic VASS. It is easy to see, that $L_1 = a^{n>0}b^{\leq n}$ is a language of a deterministic VASS. Hence it is also history-deterministic. In [7] it was shown, that history-deterministic VASS languages are closed under union. Hence $L_2 = L \cup L_1 = \{a,b\}^*a^{n>0}b^{\leq n}$ is also history-deterministic VASS language. However, in [7] it was also shown, that L_2 is not a history-deterministic VASS language. Hence we get, that L is not a history-deterministic VASS language. ◀

▶ **Lemma 17.** *There exists language* $L \in$ 1-NonDet $\setminus$ BAmb.

Proof. Let $L = \{ua^nba^mbv \mid u,v \in L((a^*b)^*), n \geq m\}$. Let us fix $k \in \mathbb{N}_+$. We show, that L is not recognised by a k-ambiguous VASS. Assume, towards contradiction, that it is. Hence language $L_k = L \cap L((a^*b)^{k+2})$ is also recognised by a k-ambiguous VASS. Let f be a function such that $f : \{\epsilon\} \to \mathbb{N}_\omega$ and $f(\epsilon) = \omega$. Therefore, by Lemma 5, we get, that L_k is not recognised by a k-ambiguous VASS, contradiction. Hence L is not recognised by a k-ambiguous VASS. On the other hand, it can be recognised by a 1-dimensional VASS, which is presented in Figure 2. ◀

Observe, that because each word is read by only finite number of accepting runs, one can see bounded-ambiguity as an extension of the notion of determinism. A second extension of the determinism is history-determinism. In [7] it was shown, that history-deterministic VASSs can express more than deterministic ones and less than nondeterministic ones. Up to now, there has been no comparison of the expressive power of history-deterministic and bounded-ambiguous VASSs. Now, we show, that these language classes are incomparable, which means that there exists a language recognised by an unambiguous VASS, which is not a language of history-deterministic VASS and language which is recognised by a history-deterministic VASS and it is not recognised by a k-ambiguous VASS for any $k \in \mathbb{N}_+$. Recall, that in Lemma 16 we have shown that there exists $L \in$ 1-Amb $\setminus$ Hist. Hence it is enough to show the following Lemma:

▶ **Lemma 18.** *There exists language* $L \in$ Hist $\setminus$ BAmb.

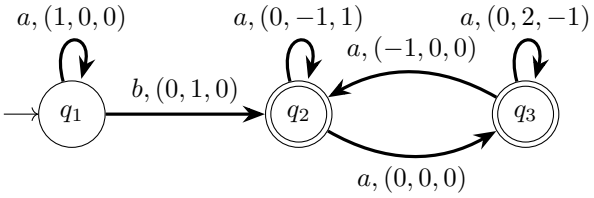


Figure 3 VASS recognising (starting from zero) $\{a^n b a^m \mid n, m \in \mathbb{N}, (m \leq 2n + 2^{n+2} - 1)\}$.

Proof. Let $L = a^n b a^{2n+2^{n+2}-1}$. Observe, that $im(L)$ is not a semilinear set. Hence, due to Lemma 8, for every $k \in \mathbb{N}_+$ we have that L is not recognised by a k-ambiguous VASS. Hence $L \notin$ BAmb. On the other hand it is recognised by a history-deterministic VASS presented in the Figure 3. It is history-deterministic, because the best option is to fire loops in states q_2 and q_3 as long as possible. In this way one can read words $a^n b a^k$ for each $k \in \mathbb{N}$ such that

$$k \leq 2 \cdot \Sigma_{i=1}^{n-1}(2^i + 1) + 2^n + 1 + 2^n = 2 \cdot (2^n - 1) + 2 \cdot n + 2^{n+1} + 1 = 2 \cdot n + 2^{n+2} - 1 \blacktriangleleft$$

5 Proof of Theorem 1

In this section we prove Theorem 1. We start by introducing *Lossy counter machines* (LCMs) [24, 31]. Formally, an LCM is $M = \langle Q, Z, \Delta \rangle$ where $Q = \{l_1, \ldots, l_m\}$ is a finite set of states, $Z = (z_1, \ldots, z_n)$ are n counters, and $\Delta \subseteq Q \times OP(Z) \times Q$, where $OP(Z) = \{\text{inc}, \text{dec}, \text{ztest}, \text{skip}\}^n$. A configuration of M is $q(a)$ where $q \in Q$ and $a = (a_1, ..., a_n) \in \mathbb{N}^n$. There is a transition $q(a) \xrightarrow{t} q'(b)$ if there exists $t \in \Delta$ such that $t = (q, op, q')$ and for each $i \in [1, n]$:

- If $op_i = \text{inc}$ then $b_i \leq a_i + 1$
- If $op_i = \text{dec}$ then $b_i \leq a_i - 1$
- If $op_i = \text{skip}$ then $b_i \leq a_i$
- If $op_i = \text{ztest}$ then $a_i = b_i = 0$

Observe, that counters can nondeterministically decrease at each step. A run of M is a finite sequence $q_1(a_1) \xrightarrow{t_1} q_2(a_2) \xrightarrow{t_2} \ldots \xrightarrow{t_{n-1}} q_n(a_n)$. Given a configuration $q(u)$, the reachability set of $q(u)$ is the set of all configurations reachable from $q(u)$ via runs of M. We denote this set as $\textsc{Reach}(q(u))$. For simplicity we denote by $\textsc{Reach}(q)$ the set of configurations reachable from $q(\vec{0})$. It was shown in [31] that the problem of deciding whether for a configuration $q(u)$ and LCM M set $\textsc{Reach}(q(u))$ configuration is finite, is undecidable. Due to [1] even if u is always equal to $\vec{0}$ the problem is still undecidable. We call this problem 0**-finite reach**. We prove Theorem 1 by reducing from 0-finite reach. The proof is similar to the proofs of undecidability of regularity [34] and determinization [1]. The rest of the section is devoted to the proof of Theorem 1.

Firstly, we present an overview of the proof. For each LCM M_1 with an initial state we create another LCM M and initial state l_0 with the same answer to the 0-finite reach problem. Then, we define a language L_{M,l_0}, which intuitively encodes the correct runs of M. For technical reasons, namely because coverability VASSs are well-suited for recognising languages similar to $a^n b^{\leq n}$ we encode the correct runs in reverse, that means from the final configuration to the initial one. Moreover, because we work with LCMs, it is better for us to work with the complement of L_{M,l_0}, that is $\overline{L_{M,l_0}}$. In such a way we get at the end a language for which Lemma 5 is useful. Then, the proof of Theorem 1 is split into three claims. Claim 19 states, that $\overline{L_{M,l_0}}$ is recognised by an effectively constructible 1-dimensional VASS

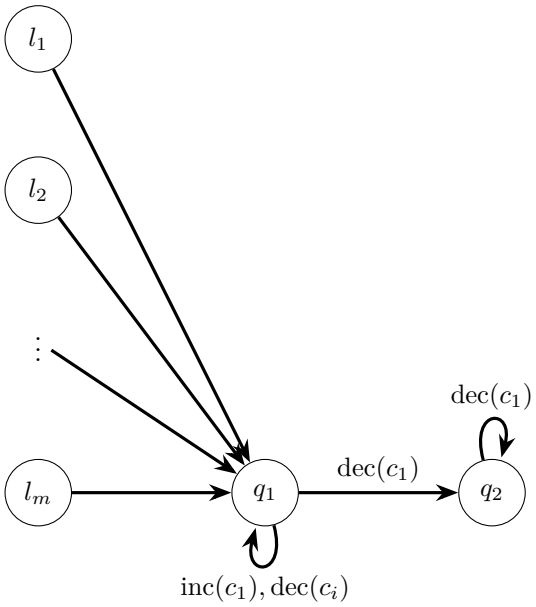


Figure 4 Sketch of the construction of LCM M.

A. Claim 20 provides, that if $\textsc{Reach}(l_0)$ is finite then L_{M,l_0} is a regular language. Finally, Claim 21 states, that if $\textsc{Reach}(l_0)$ is not finite then L_{M,l_0} is not recognised by a VASS from the class of all boundedly-ambiguous VASS. All these claims give a direct reduction from 0-finite reach to deciding whether a language of a 1-VASS belongs to class C of languages, which contains all regular languages and is contained in the class of all boundedly-ambiguous VASS languages. Thus they conclude the proof of Theorem 1.

Let us fix LCM $M_1 = \langle Q_1, Z_1, \Delta_1 \rangle$ with $Q = \{l_1, l_2, \dots l_m\}$ and $Z = \{z_1, z_2, \dots, z_n\}$. Let l_0 be the initial state of M_1. We add to M_1 two states: q_1 and q_2 and for $i \in [1, m]$ transitions t_i from l_i to q_1 with no effect on the counters. In addition, for each $i \in [2, n]$ we add a transition from q_1 to q_1 decrementing the ith counter and incrementing the first counter. We also add two transitions decrementing the first counter. The first one goes from q_1 to q_2. The second one goes from q_2 to q_2. In such a way we obtain LCM $M = \langle Q, Z, \Delta \rangle$. The sketch of the construction of M is presented in the Figure 4. Observe, that because each of the added transitions does not increase the sum of the counters, from q_1 we can go only to q_2 and later we can only stay in q_2 the answer for 0-finite reach is the same for both: M_1 and M. Hence, we proceed later with M. We encode each configuration $q(a_1, a_2, \dots, a_n)$ as a word over $\Sigma = Q \cup Z$ as $q z_1^{a_1} z_2^{a_2} \dots z_n^{a_n}$. We use encodings of configurations to obtain an encoding of a run by concatenating encodings of its configurations. Finally, we define language $L_{M,l_0} = \{w^r \mid w \text{ is an encoding of a run in } M \text{ from } l_0(0, 0, \dots, 0)\}$.

▷ Claim 19. One can construct 1-dimensional VASS recognising $\overline{L_{M,l_0}}$.

Proof. W.l.o.g. we assume, that there is at most one transition between each pair of states (otherwise we may split a state into several states, one for each incoming letter). We construct A such that it accepts w if and only if w^r does not represent a valid run of M from $l_0(0, 0, \dots, 0)$. The idea is, that A guesses the violation in the run represented by the word w. We have three types of violations. The first type is a control state violation, that is the run uses nonexisting transition or does not start in $l_0(0, 0, \dots, 0)$. The second type is a counter violation, that we have invalid counter values between two consecutive configurations.

The third violation is that we have invalid encoding of a configuration, that is we have two consecutive letters $z_i z_j$ such that $j > i$. To spot control violation for nonexisting transitions we will have gadget for each pair of states p and q such that there is no transition from p to q spotting infix of the form $z_n^* z_{n-1}^* \dots z_1^* q z_n^* z_{n-1}^* \dots z_1^* p$. Such gadget is just an NFA. Observe, that if a run start in $l_0(0,0,\dots,0)$ then either the whole encoding of the run is equal to l_0 or suffix is of the form pl_0 for some $p \in Q$. Therefore to spot control violation, that the run does not start in $l_0(0,0,\dots,0)$ we will have an NFA recognising words such that suffix is not of the form pl_0 for some $p \in Q$ and the encoding is not equal to l_0. To spot counter violation we will have a 1-dimensional VASS for each transition and each counter spotting violation when firing this transition on this counter. Let us fix transition t and counter z_i. Let transition t be from state p to state q. We have four possibilities for the operation performed by t on the counter z_i. Therefore we need to spot infix of the form:

1. $z_n^* z_{n-1}^* \dots z_i^a \dots z_1^* q z_n^* z_{n-1}^* \dots z_i^b \dots z_1^* p$ where $a > b-1$ (equivalently $a \geq b$) if transition t decrements counter z_i.
2. $z_n^* z_{n-1}^* \dots z_i^a \dots z_1^* q z_n^* z_{n-1}^* \dots z_i^b \dots z_1^* p$ where $a > b+1$ if transition t increments counter z_i.
3. $z_n^* z_{n-1}^* \dots z_i^a \dots z_1^* q z_n^* z_{n-1}^* \dots z_i^b \dots z_1^* p$ where $a > b$ if transition t has no effect on the counter z_i.
4. $z_n^* z_{n-1}^* \dots z_i^a \dots z_1^* q z_n^* z_{n-1}^* \dots z_i^b \dots z_1^* p$ where $a > 0$ or $b > 0$ if transition t zero-tests counter z_i.

All of the above can be done with 1-dimensional VASS. To spot invalid encoding of a configuration for each $1 \leq i < j \leq n$ we will have an NFA recognising words with infix $z_i z_j$. As all the gadgets have one initial state in which we "ignore" some prefix of the word, we can join them and obtain one dimensional VASS with single initial configuration. ◁

▷ Claim 20. If $\textsc{Reach}(l_0)$ is finite then $\overline{L_{M,l_0}}$ is a regular language.

Proof. Observe, that in the proof of Claim 19 only gadgets spotting counter violations were not an NFA. Therefore it is enough to replace them by some NFAs. As $\textsc{Reach}(l_0)$ is finite there exists a bound $B \in \mathbb{N}$ on the possible values of the counters. Hence we can implement each gadget spotting counter violation using the fact, that $B \geq a, b$. For instance for transition t from state p to q having no effect on the counter z_i we can have an NFA being a union of NFAs for each $0 \leq b < a \leq B$ spotting infix of the form $z_n^* z_{n-1}^* \dots z_i^a \dots z_1^* q z_n^* z_{n-1}^* \dots z_i^b \dots z_1^* p$. In this way we will not detect counter violation increasing the counter to the value above B. Therefore we add another gadget spotting, that one counter is above B hence then the word does not encode a correct run. This can be done by spotting for each $i \in [1,n]$ infix z_i^{B+1}, which can be done by an NFA. Because every gadget is now an NFA we showed that $\overline{L_{M,l_0}}$ is a regular language when $\textsc{Reach}(l_0)$ is finite. ◁

▷ Claim 21. If $\textsc{Reach}(l_0)$ is infinite then $\overline{L_{M,l_0}}$ is not recognised by a boundedly-ambiguous VASS.

Proof. Assume, towards contradiction, that $\overline{L_{M,l_0}}$ is recognized by a k-ambiguous VASS for some $k \in \mathbb{N}_+$. Recall, that there exist $q_1, q_2 \in Q$ such that from each $q_3 \in Q$ such that $q_3 \neq q_2$ and $q_3 \neq q_1$ there exists a transition from q_3 to q_1 with no effect on the counters. Moreover, for each $i \in [2,n]$ there exists a transition from q_1 to q_1 decrementing the ith counter and increasing the first counter. Additionally, there is a single transition leaving q_1 decrementing the first counter to q_2 and there is exactly one transition leaving q_2, which goes to q_2 and decrements the first counter.

As k-ambiguous VASS languages are closed under intersection with a regular language also $L_1 = \overline{L_{M,l_0}} \cap (z_1^* q_2)^{k+2} Z^* q_1 (Q \cup Z \setminus q_2)^*$ is recognised by a k-ambiguous VASS. The idea of this intersection is to get a language similar to the language presented in Lemma 5. Observe, that each $w \in L_1$ can be uniquely decomposed into $w = z_1^{n_1} q_2 z_1^{n_2} q_2 \dots z_1^{n_{k+2}} q_2 v$. We denote v by $\mathrm{suff}(w)$. We define $L = \{\mathrm{suff}(w) \mid w \in L_1\}$. We define function f on words from L by setting $f(w) = 0$ if w^r encodes an incorrect run of M from $l_0(0, 0, \dots, 0)$ and otherwise $f(w) = n$ where n is the maximal number such that $w = z_1^n v$ for some word v. Observe, that $L_1 = \{z_1^{n_1} q_2 z_1^{n_2} q_2 z_1^{n_3} q_2 \dots z_1^{n_{k+2}} q_2 w \mid w \in L, \exists_{1 \le i \le k+1} n_i \ge n_{i+1} \vee n_{k+2} \ge f(w)\}$. This is because both transition from q_1 to q_2 and loop from q_2 to q_2 decrements the first counter and therefore for each $v \in L_1$ such that $v = z_1^{n_1} q_2 z_1^{n_2} q_2 z_1^{n_3} q_2 \dots z_1^{n_{k+2}} q_2 w$ we have that v^r encodes invalid run if and only if either $\mathrm{suff}(v)^r = w^r$ encodes an incorrect run (that is $f(w) = 0 \le n_{k+2}$) or v^r encodes a correct run but $f(w) \le n_{k+2}$ (recall the definition of $f(w)$ in this case) or there exists $i \in [k+1]$ such that $n_i \ge n_{i+1}$. Moreover, notice that because $\textsc{Reach}(l_0)$ is infinite and we have transitions moving values to the first counter from all the other counters in the state q_1 for each each $B \in \mathbb{N}$ there exist $B' \in \mathbb{N}$ such that $B \le B'$ and a correct run ρ_B of M from $l_0(0, 0, \dots, 0)$ to $q_1(B', 0, \dots, 0)$. Let w_B be the encoding of this run. Observe, that $(z_1^{B'} q_2)^{k+2} w_B^r \in L_1$. Hence $w_B^r \in L$ and $f(w_B^r) = B' \ge B$. Hence $\sup(f) = \omega$ and hence, by Lemma 5, L_1 is not recognised by a k-ambiguous VASS. Contradiction. ◁

6 Deciding regularity

In this section we present a proof of the following theorem, that deciding regularity of a language of a k-ambigous VASS is decidable. This is in contrast to the general case where regularity is undecidable [34].

▶ **Theorem 22.** *For every $k \in \mathbb{N}$ it is decidable whether for a given k-ambigous VASS A language $L(A)$ is regular.*

The proof uses two, already present in the literature, results. The first result is the following Theorem from [10].

▶ **Theorem 23** (Theorem 28 [10])**.** *For any $k \in \mathbb{N}$ and a k-ambiguous VASS one can build in elementary time a downward-VASS which recognises the complement of its language.*

The second result is the decidability of regular separability of coverability VASS language and reachability VASS language. Regular separability problem asks whether for two VASSs A and B there exists a regular language L such that $L(A) \subseteq L$ and $L(B) \cap L = \emptyset$.

▶ **Theorem 24** (Theorem 7 [15])**.** *Regular separability is decidable if one VASS is a coverability VASS and the second VASS is a reachability VASS.* [2]

We devote the rest of this section to the proof of Theorem 22. Let us fix a k-ambiguous VASS V. We show how to decide regularity of a language of a k-ambiguous VASS. Using Theorem 23 we get a downward-VASS $\hat{V}$ recognising the complement of $L(V)$. Observe, that $L(V)$ is regular if and only if $L(V)$ and $L(\hat{V})$ are regular separable. Now we will use the following claim about downward-VASSs.

[2] Recently even stronger result about decidability of regular separability for reachability VASSs was proven in [23].

▷ Claim 25. For every downward-VASS one can construct reachability VASS recognising the same language.

Proof. Let us fix d-dimensional downward-VASS V and let Q be the states of V. We know, that the set of accepting configurations F is downward-closed. Hence, due to Proposition 3, we have a finite set $D \subseteq Q \times (\mathbb{N} \cup \omega)^d$ such that:

$$F = \bigcup_{q(u) \in D} \{q\} \times u\downarrow$$

We obtain reachability VASS V' recognising the same language as V by taking VASS V and for each $q(u) \in D$ adding state $q_{q(u)}$, ε transition from with no effect from q to $q_{q(u)}$. Moreover, for each $i \in [1, d]$ if $u_i \in \mathbb{N}$ we add ε transition from $q_{q(u)}$ to $q_{q(u)}$ incrementing by one the ith counter and otherwise if $u_i = \omega$ we add an ε-transition from $q_{q(u)}$ to $q_{q(u)}$ decrementing by one ith counter. Let the set of added states be equal to Q'. We set the initial configurations of V' to be the initial conditions of V. For vector $u \in (\mathbb{N} \cup \omega)^d$ let us denote by $\hat{u} \in \mathbb{N}^d$ vector such that for all $i \in [1, d]$ we have $u_i = \hat{u}_i$ if $u_i \in \mathbb{N}$ and $\hat{u}_i = 0$ otherwise. We set the accepting configurations of V' to the following set of configurations $F' = \{q_{q(u)}(\hat{u}) \mid q_{q(u)} \in Q'\}$. Now we have to prove, that $L(V) = L(V')$. For $L(V) \subseteq L(V')$ observe, that for each $w \in L(V)$ there exists configuration $q(v) \in F$ such that w is read by an accepting run ρ ending in $q(v)$. As F is a downward-closed set there is u such that $v \preceq u$ such that $q_{q(v)}(\hat{v}) \in F'$. Moreover, observe that $q_{q(v)}(\hat{v})$ is reachable from $q(u)$ in V using ε-transitions. Hence $w \in L(V')$.

For $L(V') \subseteq L(V)$ observe that for each $w \in L(V')$ we have an accepting run ρ ending in configuration $q_{q(u)}(\hat{u}) \in F'$. Due to construction of V' we know, that there exists prefix of ρ, such that it also reads w and reaches configuration $q(v)$ such that $v \preceq u$ and hence $q(v) \in F$ and $w \in L(V)$. ◁

Now we invoke Claim 25 on $\hat{V}$ to get reachability VASS $\hat{V}'$ recognising the same language. Finally, we use algorithm from Theorem 24 for V and $\hat{V}'$ knowing that they are regular separable if and only if $L(V)$ is regular. This concludes the proof of Theorem 22.

7 Future research

Other problems for boundedly-ambiguous VASSs. We have shown in Section 6 that the regularity problem is decidable for baVASSs (boundedly-ambiguous VASSs), in contrast to general VASSs. It is also known that the language equivalence problem is decidable for baVASSs [11], while being undecidable for general VASSs [2, 21]. It is natural to ask whether other classical problems are decidable for baVASSs, for example deciding whether there exists an equivalent deterministic or unambiguous VASS. These problems are undecidable for general VASS due to Theorem 1, but can possibly be decidable for baVASSs. One can also ask whether given k-ambiguous VASS has an equivalent $(k-1)$-ambiguous VASS, our techniques from Section 5 used for showing undecidability does not seem to work in that case. Other further research for baVASSs would be to ask about complexity of the mentioned problems, for example to understand complexity of the regularity problem for baVASSs, since we already know it is decidable.

Languages of VASSs accepting by configuration. In this paper we have considered VASSs accepting by set of accepting states, it is natural to ask what happens if we modify the acceptance set to be a finite set of accepting configurations or even a single configuration. For

VASSs accepting by configuration already the language universality problem is undecidable. We are not aware of any citation, but the proof is rather straightforward and uses the classical technique. For a given two-counter automaton one can construct 1-VASS accepting by configuration, which recognises all the words beside correct encodings of the runs of the two-counter automaton. Therefore the reachability problem for two-counter automaton, which is undecidable, can be reduced to the universality problem for 1-VASSs accepting by a configuration. Since the universality problem is undecidable for VASSs accepting by configuration, there is not much hope that the other nontrivial problems (beside emptiness) are decidable.

However, one can ask what about VASSs accepting by configuration with some restriction on nondeterminism, for example unambiguous or boundedly-ambiguous VASSs accepting by configuration. It is natural to ask whether universality, language equivalence, regularity or determinisability problems are decidable for that models and what is its complexity.

References

1 Shaull Almagor and Asaf Yeshurun. Determinization of One-Counter Nets. In *Proceedings of CONCUR 2022*, volume 243 of *LIPIcs*, pages 18:1–18:23. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICS.CONCUR.2022.18.

2 Toshiro Araki and Tadao Kasami. Some Decision Problems Related to the Reachability Problem for Petri Nets. *Theor. Comput. Sci.*, 3(1):85–104, 1976. doi:10.1016/0304-3975(76)90067-0.

3 Corentin Barloy and Lorenzo Clemente. Bidimensional linear recursive sequences and universality of unambiguous register automata. In *Proceedings of STACS 2021)*, volume 187 of *LIPIcs*, pages 8:1–8:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICS.STACS.2021.8.

4 Jason P. Bell and Daniel Smertnig. Computing the linear hull: Deciding Deterministic? and Unambiguous? for weighted automata over fields. In *Proceedings of LICS 2023*, pages 1–13. IEEE, 2023. doi:10.1109/LICS56636.2023.10175691.

5 Mikolaj Bojanczyk, Joanna Fijalkow, Bartek Klin, and Joshua Moerman. Orbit-Finite-Dimensional Vector Spaces and Weighted Register Automata. *TheoretiCS*, 3, 2024. doi:10.46298/THEORETICS.24.13.

6 Mikolaj Bojanczyk, Bartek Klin, and Joshua Moerman. Orbit-finite-dimensional vector spaces and weighted register automata. In *Proceedings of LICS 2021*, pages 1–13. IEEE, 2021. doi:10.1109/LICS52264.2021.9470634.

7 Sougata Bose, David Purser, and Patrick Totzke. History-Deterministic Vector Addition Systems. In *Proceedings of CONCUR 2023*, volume 279 of *LIPIcs*, pages 18:1–18:17, 2023. doi:10.4230/LIPICS.CONCUR.2023.18.

8 Thomas Colcombet. Unambiguity in automata theory. In *Proceedings of DCFS 2015*, pages 3–18, 2015. doi:10.1007/978-3-319-19225-3_1.

9 Wojciech Czerwinski, Diego Figueira, and Piotr Hofman. Universality Problem for Unambiguous VASS. In *Proceedings of CONCUR 2020*, volume 171 of *LIPIcs*, pages 36:1–36:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPICS.CONCUR.2020.36.

10 Wojciech Czerwinski and Piotr Hofman. Language Inclusion for Boundedly-Ambiguous Vector Addition Systems Is Decidable. In *Proceedings of CONCUR 2022*, volume 243 of *LIPIcs*, pages 16:1–16:22. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICS.CONCUR.2022.16.

11 Wojciech Czerwinski and Piotr Hofman. Language Inclusion for Boundedly-Ambiguous Vector Addition Systems is Decidable. *Log. Methods Comput. Sci.*, 21(1), 2025. doi:10.46298/LMCS-21(1:27)2025.

12 Wojciech Czerwinski and Slawomir Lasota. Regular Separability of One Counter Automata. *Log. Methods Comput. Sci.*, 15(2), 2019. doi:10.23638/LMCS-15(2:20)2019.

13 Wojciech Czerwiński, Sławomir Lasota, Roland Meyer, Sebastian Muskalla, K. Narayan Kumar, and Prakash Saivasan. Regular separability of well-structured transition systems. In *29th International Conference on Concurrency Theory, CONCUR 2018*, volume 118 of *LIPIcs*, pages 35:1–35:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.CONCUR.2018.35.

14 Wojciech Czerwinski, Antoine Mottet, and Karin Quaas. New techniques for universality in unambiguous register automata. In *Proceedings of ICALP 2021*, volume 198 of *LIPIcs*, pages 129:1–129:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICS.ICALP.2021.129.

15 Wojciech Czerwinski and Georg Zetzsche. An Approach to Regular Separability in Vector Addition Systems. In *LICS '20: 35th Annual ACM/IEEE Symposium on Logic in Computer Science, Saarbrücken, Germany, July 8-11, 2020*, pages 341–354. ACM, 2020. doi:10.1145/3373718.3394776.

16 L.E. Dickson. Finiteness of the odd perfect and primitive abundant numbers with n distinct prime factors. *American Journal of Mathematics*, 35((4)):413–422, 1913.

17 Seymour Ginsburg and Joseph Ullian. Ambiguity in Context Free Languages. *J. ACM*, 13(1):62–89, 1966. doi:10.1145/321312.321318.

18 Mika Göös, Stefan Kiefer, and Weiqiang Yuan. Lower Bounds for Unambiguous Automata via Communication Complexity. In *Proceedings of ICALP 2022*, volume 229 of *LIPIcs*, pages 126:1–126:13. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICS.ICALP.2022.126.

19 Sheila A. Greibach. A Note on Undecidable Properties of Formal Languages. *Math. Syst. Theory*, 2(1):1–6, 1968. doi:10.1007/BF01691341.

20 Christoph Haase. A survival Guide to Presburger Arithmetic. *ACM SIGLOG News*, 5(3):67–82, 2018. doi:10.1145/3242953.3242964.

21 Piotr Hofman, Richard Mayr, and Patrick Totzke. Decidability of weak simulation on one-counter nets. In *Proceedings of LICS 2013*, pages 203–212. IEEE Computer Society, 2013. doi:10.1109/LICS.2013.26.

22 Richard M. Karp and Raymond E. Miller. Parallel program schemata. *J. Comput. Syst. Sci.*, 3(2):147–195, 1969. doi:10.1016/S0022-0000(69)80011-5.

23 Eren Keskin and Roland Meyer. On the separability problem of VASS reachability languages. In *Proceedings of the 39th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2024, Tallinn, Estonia, July 8-11, 2024*, pages 49:1–49:14. ACM, 2024. doi:10.1145/3661814.3662116.

24 Richard Mayr. Undecidable problems in unreliable computations. *Theor. Comput. Sci.*, 297(1-3):337–354, 2003. doi:10.1016/S0304-3975(02)00646-1.

25 Antoine Mottet and Karin Quaas. The containment problem for unambiguous register automata. In *Proceedings of STACS 2019*, pages 53:1–53:15, 2019. doi:10.4230/LIPICS.STACS.2019.53.

26 Frank Neven, Thomas Schwentick, and Victor Vianu. Finite state machines for strings over infinite alphabets. *ACM Trans. Comput. Log.*, 5(3):403–435, 2004. doi:10.1145/1013560.1013562.

27 Aditya Prakash and K. S. Thejaswini. On History-Deterministic One-Counter Nets. In *Proceedings of FoSSaCS 2023*, volume 13992 of *Lecture Notes in Computer Science*, pages 218–239. Springer, 2023. doi:10.1007/978-3-031-30829-1_11.

28 Antoni Puch and Daniel Smertnig. Factoring through monomial representations: arithmetic characterizations and ambiguity of weighted automata. *CoRR*, abs/2410.03444, 2024. doi:10.48550/arXiv.2410.03444.

29 Mikhail Raskin. A superpolynomial lower bound for the size of non-deterministic complement of an unambiguous automaton. In *Proceedings of ICALP 2018*, pages 138:1–138:11, 2018. doi:10.4230/LIPICS.ICALP.2018.138.

30 Sylvain Schmitz. The complexity of reachability in vector addition systems. *ACM SIGLOG News*, 3(1):4–21, 2016. doi:10.1145/2893582.2893585.

31 Philippe Schnoebelen. Lossy Counter Machines Decidability Cheat Sheet. In *Reachability Problems, 4th International Workshop, RP 2010, Brno, Czech Republic, August 28-29, 2010. Proceedings*, volume 6227 of *Lecture Notes in Computer Science*, pages 51–75. Springer, 2010. doi:10.1007/978-3-642-15349-5_4.

32 Richard Edwin Stearns and Harry B. Hunt III. On the equivalence and containment problems for unambiguous regular expressions, regular grammars and finite automata. *SIAM J. Comput.*, 14(3):598–611, 1985. doi:10.1137/0214044.

33 Wen-Guey Tzeng. On path equivalence of nondeterministic finite automata. *Inf. Process. Lett.*, 58(1):43–46, 1996. doi:10.1016/0020-0190(96)00039-7.

34 Rüdiger Valk and Guy Vidal-Naquet. Petri nets and regular languages. *J. Comput. Syst. Sci.*, 23(3):299–325, 1981. doi:10.1016/0022-0000(81)90067-2.

35 Łukasz Orlikowski. Languages of Unambiguous Vector Addition Systems with States. Master's thesis, University of Warsaw, 2024. URL: https://apd.uw.edu.pl/diplomas/225085/.

A Proofs for Section 3 (Tools for separating BAmb and NonDet)

Proof of Lemma 7. We prove this lemma by induction on n. For $n = 0$ we have $\rho = \varepsilon$ and hence all decomposition conditions are satisfied. Now we assume, that this Lemma is true for $n-1$ and we show that this implies Lemma 7 for n.

Let π_1 be the prefix of ρ reading $a^{m_1}ba^{m_2}b\dots a^{m_{n-1}}$ and let t be the next transition of ρ after π_1. Hence $\rho = \pi_1 t \pi_2$ for some π_2. By induction assumption we can apply Lemma 7 to π_1 and get constant C and decomposition:

$$\pi_1 = \alpha_1\beta_1^{a_1}\alpha_1'\alpha_2\beta_2^{a_2}\alpha_2'\dots\alpha_{n-1}\beta_{n-1}^{a_{n-1}}\lambda$$

Observe, that we changed α_{n-1}' to λ as we will redefine α_{n-1}'. Now let us set $\alpha_{n-1}' = \lambda t$. Hence

$$\rho = \alpha_1\beta_1^{a_1}\alpha_1'\alpha_2\beta_2^{a_2}\alpha_2'\dots\alpha_{n-1}\beta_{n-1}^{a_{n-1}}\alpha_{n-1}'\pi_2$$

Recall the definition of set A_n from condition 5, let m be the maximal norm of a transition, s be the maximal norm of an initial configuration and let us define constants $T = s + 2(n-1)mC$ and $C_n = M(V, A_n, T)$ given by Lemma 6. Let us also define constant $K_n = \max_{S\subseteq[d]} M(V, S, T + mC_n)$. Let also $C' = \max(C+1, C_n, K_n)$. This will be a constant for Lemma 7 and n. We have two cases. The first case is that $\pi_2 = \alpha_n\beta_n^{a_n}\alpha_n'$ for some loop β_n, which is nonnegative on counters from $[d]\setminus A_n$ and $|\alpha_n\beta_n| \le C_n$ and β_n is not a prefix of α_n'. The second case is that such a decomposition of π_2 does not exist. We start the proof with the second case.

Case 2. Observe, that for each $i \in [n-1]$ and $l \in [d]\setminus A_n \subseteq [d]\setminus A_i$:

$$\textsc{eff}_l(\beta_i) = 0$$

Hence:

$$\textsc{eff}_l(\alpha_1\beta_1^{a_1}\alpha_1'\alpha_2\beta_2^{a_2}\alpha_2'\dots\alpha_{n-1}\beta_{n-1}^{a_{n-1}}\alpha_{n-1}') = \Sigma_{i=1}^{n-1}(\textsc{eff}_l(\alpha_i) + \textsc{eff}_l(\alpha_i')) \le m\Sigma_{i=1}^{n-1}(|\alpha_i| + |\alpha_i'|) \le 2(n-1)mC = T$$

Hence, because of the definition of C_n and the fact that in this case, the decomposition of π_2 does not exist, we have that $|\pi_2| \le C_n \le C'$. Then we set $\alpha_n = \pi_2$ and $\beta_n = \alpha_n' = \varepsilon$. Then clearly from the induction assumption and the definition of $\alpha_{n-1}', \alpha_n, \beta_n, \alpha_n'$ conditions 1-3 and 5 are satisfied. Condition 4 is satisfied for $j < n$ by induction assumption and for $j = n$ we have $m_j = |\alpha_n\beta_n\alpha_n'| \le C_n \le C$ hence it is also satisfied. Condition 6 is satisfied for $j < n$ by induction assumption. It is satisfied for $j = n$ because otherwise π_2 could have been decomposed and we are in case 1.

Case 1. We know, that π_2 can be decomposed as described above. If there are multiple decompositions we choose any of the ones minimizing $|\alpha_n\beta_n|$. Now, we show that decomposition of ρ:

$$\rho = \alpha_1\beta_1^{a_1}\alpha_1'\alpha_2\beta_2^{a_2}\alpha_2'\dots\alpha_n\beta_n^{a_n}\alpha_n'$$

satisfies all the conditions. We start with conditions 2-6 as Condition 1 is the most challenging to prove.

Condition 2. For $j < n-1$ this condition follows from the induction assumption. For $j = n-1$ we have that $\alpha_{n-1}' = \lambda t$. Recall that t was a transition reading letter b and from the induction assumption λ does not read letter b. Moreover, α_n' is a suffix of π_2, which does not read the letter b. Hence the condition follows.

Condition 3. For $j < n$ the condition follows from the induction assumption and for $j = n$ we have that $\alpha_n\beta_n$ is a prefix of π_2, which does not read the letter b.

Condition 4. For $j < n$ we have this condition from induction assumption and for $j = n$ we have that $\beta_n \neq \varepsilon$ is a loop.

Condition 5. This condition for $j < n$ follows from inductions assumption and for $j = n$ follows from the fact that β_n is a nonnegative loop on counters from $[d] \setminus A_n$ as required.

Condition 6. For $j < n$ we have it from induction assumption. For $j = n$ we have it from minimality of $|\alpha_n\beta_n|$.

Condition 1. Observe, that for $i \in [n-2]$ from induction assumption we have $|\alpha_i|, |\beta_i|, |\alpha_i'| \leq C \leq C'$ and moreover $|\alpha_{n-1}|, |\beta_{n-1}| \leq C \leq C'$. Moreover, from induction assumption, $|\alpha_{n-1}|, |\beta_{n-1}| \leq C \leq C'$ and $|\alpha_{n-1}'| = |\lambda t| \leq C+1 \leq C'$. Additionally $|\alpha_n\beta_n| \leq C_n \leq C'$. Therefore it is enough to prove that $|\alpha_n'| \leq K_n \leq C'$. Let

$$S = [d] \setminus \bigcup_{i=1}^{n} \text{support}(\beta_i)$$

Observe, that for each $l \in S$ we have:

$$\text{EFF}_l(\alpha_1\beta_1^{a_1}\alpha_1'\alpha_2\beta_2^{a_2}\alpha_2'\dots\alpha_n\beta_n^{a_n}) = \text{EFF}_l(\alpha_n) + \Sigma_{i=1}^{n-1}(\text{EFF}_l(\alpha_i) + \text{EFF}_l(\alpha_i')) \leq mC_n + 2(n-1)mC$$

Recall that $K_n = M(V, S, T + mC_n)$ and $T = 2(n-1)mC$, so if $|\alpha_n'| > K_n$ then α_n' contains a nonnegative loop on counters from S. Hence it is enough to prove, that α_n' does not contain such loop. Assume, for the sake of contradiction, that $\alpha_n' = \lambda\gamma\lambda'$ such that γ is a nonnegative loop on counters from S. We define accepting runs $\phi_1, \phi_2, \dots, \phi_{k+1}$, which will read the same word, but will be different and hence this will create a contradiction with the fact that V is k-ambiguous VASS. Intuitively, due to loops β_n and γ we have more degrees of freedom in the last block and we can create $k+1$ different runs over the same word. As ρ is a prefix of an accepting run we know, that there exists ρ_1 such that $\rho\rho_1$ is an accepting run. Firstly, we define runs ψ_1, ψ_2 and ψ_3, which will be parts of $\phi_1, \phi_2, \dots, \phi_{k+1}$:

$$\psi_1 = \alpha_1\beta_1^{a_1+b_1}\alpha_1'\alpha_2\beta_2^{a_2+b_2}\alpha_2'\dots\alpha_n\beta_n^{a_n+b_n}$$

$$\psi_2 = \lambda\gamma$$

$$\psi_3 = \lambda'\rho_1$$

where for $i \in [n]$ we define $b_i = (k+1)m|\gamma||\beta_n| + \Sigma_{j=i+1}^{n} b_j m|\beta_j|$. The intuition is, that b_i is chosen in such a way to compensate the possible decrease of counters because of additional executions of loops $\beta_{i+1}, \beta_{i+2}, \ldots, \beta_n$ and γ. Finally we define ϕ_i

$$\phi_i = \psi_1 \beta_n^{i|\gamma|} \psi_2 \gamma^{(k+1-i)|\beta_n|} \psi_3$$

where for $i \in [n]$ we define $b_i = (k+1)m|\gamma||\beta_n| + \Sigma_{j=i+1}^{n} b_j m|\beta_j|$. Now it is enough to prove Claims 26, 27 and 28.

▷ Claim 26. For each $i \in [k+1]$ we have that ϕ_i is an accepting run.

Proof. Let us fix $i \in [k+1]$. Because $\rho\rho_1$ is an accepting run it is enough to prove that:

1. For each $j \in [n-1]$ and $l \in [d]$ we have

$$\textsc{eff}_l(\alpha_1 \beta_1^{a_1+b_1} \alpha_1' \alpha_2 \beta_2^{a_2+b_2} \alpha_2' \ldots \alpha_j \beta_j^{a_j+b_j}) \geq \textsc{eff}_l(\alpha_1 \beta_1^{a_1} \alpha_1' \alpha_2 \beta_2^{a_2} \alpha_2' \ldots \alpha_j \beta_j^{a_j})$$

2. For each $l \in [d]$ we have

$$\textsc{eff}_l(\psi_1 \beta_n^{i|\gamma|}) \geq \textsc{eff}_l(\alpha_1 \beta_1^{a_1} \alpha_1' \alpha_2 \beta_2^{a_2} \alpha_2' \ldots \alpha_n \beta_n^{a_n})$$

3. For each $l \in [d]$ we have

$$\textsc{eff}_l(\psi_1 \beta_n^{b_n+i|\gamma|} \psi_2 \gamma^{(k+1-i)|\beta_n|}) \geq \textsc{eff}_l(\alpha_1 \beta_1^{a_1} \alpha_1' \alpha_2 \beta_2^{a_2} \alpha_2' \ldots \alpha_n \beta_n^{a_n} \lambda\gamma)$$

Point 1. We prove this by induction on j. For $j = 1$ the inequality follows from the fact that for each $l \in [d]$ we have $\textsc{eff}_l(\beta_1) \geq 0$. Now we show the induction step. If $\textsc{eff}_l(\beta_j) \geq 0$ we get the inequality from the induction assumption. Hence now we assume $\textsc{eff}_l(\beta_j) < 0$. Therefore there has to be $u < j$ such that $\textsc{eff}_l(\beta_u) > 0$. Let u be the maximal such. Observe, that from induction assumption:

$$\textsc{eff}_l(\alpha_1 \beta_1^{a_1+b_1} \alpha_1' \alpha_2 \beta_2^{a_2+b_2} \alpha_2' \ldots \alpha_{s-1} \beta_{u-1}^{a_{u-1}+b_{u-1}} \alpha_u \beta_u^{a_u} \alpha_u' \ldots \alpha_j \beta_j^{a_j} \alpha_j') \geq \textsc{eff}_l(\alpha_1 \beta_1^{a_1} \alpha_1' \alpha_2 \beta_2^{a_2} \alpha_2' \ldots \alpha_j \beta_j^{a_j} \alpha_j')$$

And moreover:

$$b_u \textsc{eff}_l(\beta_u) \geq b_u$$

and for $t \in [u+1, j]$ we have:

$$b_t \textsc{eff}_l(\beta_t) \geq -b_t m|\beta_t|$$

Hence:

$$\begin{aligned}
&\textsc{eff}_l(\alpha_1 \beta_1^{a_1+b_1} \alpha_1' \alpha_2 \beta_2^{a_2+b_2} \alpha_2' \ldots \alpha_j \beta_j^{a_j+b_j}) = \\
&\textsc{eff}_l(\alpha_1 \beta_1^{a_1+b_1} \alpha_1' \alpha_2 \beta_2^{a_2+b_2} \alpha_2' \ldots \alpha_{s-1} \beta_{u-1}^{a_{u-1}+b_{u-1}} \alpha_u \beta_u^{a_u} \alpha_u' \ldots \alpha_j \beta_j^{a_j} \alpha_j') + \Sigma_{t=u}^{j} b_t \textsc{eff}_l(\beta_t) \geq \\
&\textsc{eff}_l(\alpha_1 \beta_1^{a_1} \alpha_1' \alpha_2 \beta_2^{a_2} \alpha_2' \ldots \alpha_j \beta_j^{a_j} \alpha_j') + b_u - \Sigma_{t=u+1}^{j} b_t m|\beta_t| \geq \textsc{eff}_l(\alpha_1 \beta_1^{a_1} \alpha_1' \alpha_2 \beta_2^{a_2} \alpha_2' \ldots \alpha_j \beta_j^{a_j} \alpha_j')
\end{aligned}$$

Point 2. If $\text{EFF}_l(\beta_n) \geq 0$ than because of point 1 we have:

$$\text{EFF}_l(\alpha_1\beta_1^{a_1+b_1}\alpha_1'\alpha_2\beta_2^{a_2+b_2}\alpha_2'\dots\alpha_n\beta_n^{a_n+b_n+i|\gamma|}) \geq \text{EFF}_l(\alpha_1\beta_1^{a_1}\alpha_1'\alpha_2\beta_2^{a_2}\alpha_2'\dots\alpha_n\beta_n^{a_n})$$

Hence, we can assume $\text{EFF}_l(\beta_n) < 0$. Therefore there has to be $j < n$ such that $\text{EFF}_l(\beta_j) > 0$. Let j be the maximal such. Observe, that:

$$\text{EFF}_l(\alpha_1\beta_1^{a_1+b_1}\alpha_1'\alpha_2\beta_2^{a_2+b_2}\alpha_2'\dots\alpha_j\beta_j^{a_j}\alpha_j'\beta_{j+1}^{a_{j+1}}\alpha_{j+1}'\dots\alpha_n\beta_n^{a_n}) \geq \text{EFF}_l(\alpha_1\beta_1^{a_1}\alpha_1'\alpha_2\beta_2^{a_2}\alpha_2'\dots\alpha_n\beta_n^{a_n})$$

$$b_j\text{EFF}_l(\beta_j) \geq b_j$$

And for each $s \in [j+1, n]$

$$b_s\text{EFF}_l(\beta_s) \geq b_s m|\beta_s|$$

Hence:

$$\begin{aligned}&\text{EFF}_l(\alpha_1\beta_1^{a_1+b_1}\alpha_1'\alpha_2\beta_2^{a_2+b_2}\alpha_2'\dots\alpha_n\beta_n^{a_n+b_n+i|\gamma|}) = \\ &\text{EFF}_l(\alpha_1\beta_1^{a_1+b_1}\alpha_1'\alpha_2\beta_2^{a_2+b_2}\alpha_2'\dots\alpha_j\beta_j^{a_j}\alpha_j'\beta_{j+1}^{a_{j+1}}\alpha_{j+1}'\dots\alpha_n\beta_n^{a_n}) + \Sigma_{s=j}^n b_s\text{EFF}_l(\beta_s) + i|\gamma|\text{EFF}_l(\beta_n) \geq \\ &\text{EFF}_l(\alpha_1\beta_1^{a_1}\alpha_1'\alpha_2\beta_2^{a_2}\alpha_2'\dots\alpha_n\beta_n^{a_n}) + b_j - \Sigma_{s=j+1}^n b_s m|\beta_s| - mi|\lambda||\beta_n| \geq \text{EFF}_l(\alpha_1\beta_1^{a_1}\alpha_1'\alpha_2\beta_2^{a_2}\alpha_2'\dots\alpha_n\beta_n^{a_n})\end{aligned}$$

Point 3. If $\text{EFF}_l(\gamma) \geq 0$ than the inequality follows from Point 2. Hence now we assume $\text{EFF}_l(\gamma) < 0$. Therefore there has to be $j \in [n]$ such that $\text{EFF}_l(\beta_j) > 0$. Let j be the maximal such. Hence: Observe, that:

$$\text{EFF}_l(\alpha_1\beta_1^{a_1+b_1}\alpha_1'\alpha_2\beta_2^{a_2+b_2}\alpha_2'\dots\alpha_j\beta_j^{a_j}\alpha_j'\dots\alpha_n\beta_n^{a_n}\lambda\gamma) \geq \text{EFF}_l(\alpha_1\beta_1^{a_1}\alpha_1'\alpha_2\beta_2^{a_2}\alpha_2'\dots\alpha_j\beta_j^{a_j}\alpha_j'\dots\alpha_n\beta_n^{a_n}\lambda\gamma)$$

$$b_j\text{EFF}_l(\beta_j) \geq b_j$$

And for $s \in [j+1, n]$ we have:

$$b_s\text{EFF}_l(\beta_s) \geq b_s m|\beta_s|$$

$$\begin{aligned}&\text{EFF}_l(\alpha_1\beta_1^{a_1+b_1}\alpha_1'\alpha_2\beta_2^{a_2+b_2}\alpha_2'\dots\alpha_n\beta_n^{a_n+b_n+i|\gamma|}\lambda\gamma^{1+(k+1-i)|\beta_n|}) = \\ &\text{EFF}_l(\alpha_1\beta_1^{a_1+b_1}\alpha_1'\alpha_2\beta_2^{a_2+b_2}\alpha_2'\dots\alpha_j\beta_j^{a_j}\alpha_j'\dots\alpha_n\beta_n^{a_n}\lambda\gamma) + b_j\text{EFF}_l(\beta_j) + \\ &\Sigma_{s=j+1}^n b_s\text{EFF}_l(\beta_s) + i|\gamma|\text{EFF}_l(\beta_n) + (k+1-i)|\beta_n|\text{EFF}_l(\gamma) \geq \\ &\text{EFF}_l(\alpha_1\beta_1^{a_1}\alpha_1'\alpha_2\beta_2^{a_2}\alpha_2'\dots\alpha_j\beta_j^{a_j}\alpha_j'\dots\alpha_n\beta_n^{a_n}\lambda\gamma) + b_j - \Sigma_{s=j+1}^n b_s m|\beta_s| - (k+1)m|\beta_n||\gamma| = \\ &\qquad\qquad \text{EFF}_l(\alpha_1\beta_1^{a_1}\alpha_1'\alpha_2\beta_2^{a_2}\alpha_2'\dots\alpha_j\beta_j^{a_j}\alpha_j'\dots\alpha_n\beta_n^{a_n}\lambda\gamma)\end{aligned}$$

◁

▷ Claim 27. For each $i \in [k+1]$ run ϕ_i reads the same word.

Proof. Let us fix $i, j \in [k+1]$ such that $i < j$. Observe, that $w(\phi_i) = w(\phi_j)$ if and only if $w(\beta_n^{(j-i)|\gamma|}\lambda)$ and $w(\lambda\gamma^{(j-i)|\beta_n|})$. We have, that $w(\beta_n), w(\alpha_n') \in L(a^*)$. Hence also $\gamma, \lambda \in L(a^*)$, as they are parts of α_n'. Therefore these two runs read the same word because

$$|\beta_n^{(j-i)|\gamma|}\lambda| = |\lambda\gamma^{(j-i)|\beta_n|}|$$

◁

▷ Claim 28. For each $i, j \in [k+1]$ such that $i \neq j$ we have that $\phi_i \neq \phi_j$.

Proof. W.l.o.g. assume $i < j$. Observe, that it is enough to show, that

$$\beta_n^{(j-i)|\gamma|}\lambda \neq \lambda\gamma^{(j-i)|\beta_n|}$$

Assume, for the sake of contradiction that:

$$\beta_n^{(j-i)|\gamma|}\lambda = \lambda\gamma^{(j-i)|\beta_n|}$$

Observe, that β_n is not a prefix of $\lambda\gamma$, because then β_n would be a prefix of α'_n, which is not possible because of properties of π_2 (recall, that $\pi_2 = \alpha_n\beta_n^{a_n}\alpha'_n$ and β_n is not a prefix of α'_n). Hence $\lambda\gamma$ is a prefix of β_n.

Hence for all $l \in [d]$ we have:

$$\textsc{eff}_l(\beta_n^{(j-i)|\gamma|}\lambda) = \textsc{eff}_l(\lambda\gamma^{(j-i)|\beta_n|})$$

Hence $\text{support}(\beta_n) = \text{support}(\gamma)$. Because $\lambda\gamma$ is a prefix of β_n and $\lambda\gamma \neq \beta_n$ this contradicts the minimality of $|\alpha_n\beta_n|$. ◁

Since we have proven Claims 26, 27 and 28 we know that runs ϕ_i are all accepting, different and read the same word. This concludes the proof of Lemma 7. ◀

Reversible Pebble Transducers

Luc Dartois
Université Paris Est Creteil, LACL, F-94010 Créteil, France

Paul Gastin
Université Paris-Saclay, ENS Paris-Saclay, CNRS, LMF, 91190, Gif-sur-Yvette, France
CNRS, ReLaX, IRL 2000, Siruseri, India

Loïc Germerie Guizouarn
Univ Rennes, CNRS, Inria, IRISA - UMR 6074, F-35000 Rennes, France

Shankaranarayanan Krishna
Indian Institute of Technology Bombay, Mumbai, India

Abstract

Deterministic two-way transducers with pebbles (aka pebble transducers) capture the class of polyregular functions, which extend the string-to-string regular functions allowing polynomial growth instead of linear growth. One of the most fundamental operations on functions is composition, and (poly)regular functions can be realized as a composition of several simpler functions. In general, composition of deterministic two-way transducers incur a doubly exponential blow-up in the size of the inputs. A major improvement in this direction comes from the fundamental result of Dartois et al. [10] showing a polynomial construction for the composition of *reversible* two-way transducers.

A precise complexity analysis for existing composition techniques of pebble transducers is missing. But they rely on the classic composition of two-way transducers and inherit the double exponential complexity. To overcome this problem, we introduce *reversible* pebble transducers. Our main results are efficient uniformization techniques for non-deterministic pebble transducers to reversible ones and efficient composition for reversible pebble transducers.

2012 ACM Subject Classification Theory of computation → Transducers

Keywords and phrases Transducers, Polyregular functions, Reversibility, Composition, Uniformization

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2025.14

Related Version *Full Version*: https://arxiv.org/abs/2506.11334

1 Introduction

The theory of string transformations has garnered a lot of attention in recent years. The simplest kind of such transformations are *sequential functions* which are captured by deterministic finite automata whose transitions are labelled by output words. An example of a sequential function is $f(w) = w'$ where w, w' are words over an alphabet $\{a, b\}$, and w' is obtained from w by replacing in w, each a with bb, and each b with a. *Rational functions* strictly extend sequential functions by allowing the underlying automaton to be *unambiguous* instead of being deterministic. For instance, $f(ua) = au$ where $a \in \Sigma, u \in \Sigma^+$ which moves the last symbol of the input to the first position is a rational function, and not realizable by a sequential one.

While both sequential and rational functions are captured by "one-way" transducers, *regular functions* are realized by two-way transducers where the input is scanned in both directions. Regular functions strictly extend rational functions; for instance the regular function $f(w) = w^R$ where w^R is the reverse of w is not realizable as a rational function. Regular functions are also equivalent to Courcelle's word-to-word MSO transductions [9, 14], streaming string transducers [1] as well as combinator expressions [2, 3, 6, 12].

© Luc Dartois, Paul Gastin, Loïc Germerie Guizouarn, and Shankaranarayanan Krishna;
licensed under Creative Commons License CC-BY 4.0

Sequential, rational as well as regular functions are all transformations of *linear growth*, where the sizes of the outputs are linear in the sizes of the inputs. More recently, [5] revisits functions of *polynomial growth*, that is, those whose output sizes are polynomial in the sizes of the input. These functions, aptly called *polyregular functions*, date back to [13, 15]. A logical characterization for polyregular functions was given in [7], namely, word-to-word MSO interpretations. MSO interpretations are equivalent to polyregular functions which are word-to-word functions recognised by deterministic, two-way pebble transducers [4].

Prior work [13, 4] on pebble transducers show that they are closed under composition, and are equivalent to polyregular functions. [13] of pebble transducers, as well as a *uniformization* of (non-determinsitic) pebble to deterministic ones with the same number of pebbles. The pebble transducers of [4] allow *comparison tests*, namely, one can compare the positions on which pebbles are dropped, or the position of the head and of a pebble. [4] gives high level ideas for the closure under composition for pebble transducers with comparison tests.

Our focus in this paper is on the complexity aspects of the composition and uniformization of pebble transducers. A practical setting where composition is useful is while synthesizing pebble transducers from polyregular functions [7]. Polyregular functions [5] are defined as the smallest class of functions closed under composition that contains sequential functions, the squaring function as well as the iterated reverse. In general, the composition of (pebble-less) two-way transducers has at least a doubly exponential blowup in the size of the input transducers [8]. An important class of two-way transducers for which composition is polynomial are *reversible* transducers [10]. Reversible transducers are those which are both deterministic and reverse-deterministic; moreover they are expressively equivalent to two-way transducers [10]. This makes reversibility a very attractive property for a transducer. However, the notion of reversible pebble transducers has not been considered yet. Therefore it is unknown if they have the same expressiveness as deterministic pebble transducers (polyregular functions), and whether they are also amenable to an efficient composition like their pebble-less counterparts. Our paper fills this gap.

Our Contributions

1. **Reversible Pebble Transducers**. We define reversible pebble transducers with equality tests allowing to check whether two pebbles are dropped on the same position, in addition to the basic tests allowing to check if a pebble is dropped on the current head position.
2. **Equivalent notions for pebble transducers**. We show that pebble transducers with basic tests only, as well as those extended with equality tests are equivalent. More precisely, given a k-pebble transducer with equality tests having n states, we show that we can construct an equivalent k-pebble transducer with basic tests having $\mathcal{O}(nk^2)$ states. The construction preserves both determinism and reverse-determinism.
3. **Composition of Reversible Pebble Transducers**. We show that reversible pebble transducers are closed under composition. Given points 1,2 above, we allow equality tests while studying composition. To be precise, given reversible pebble transducers $\mathcal{T}$ and $\mathcal{T}'$ with states Q, Q', having $n, m \geq 0$ pebbles, we construct a reversible pebble transducer $\mathcal{T}''$ with $nm + n + m$ pebbles and (i) $2 \cdot |Q| \cdot |Q'|$ states when $m = 0$, and (ii) $\mathcal{O}(|Q|^{m+2} \cdot |Q'| \cdot (n+1)^{m+3})$ states when $m > 0$.
   To compare with the existing composition approaches [13] and [4], the number of pebbles we require for composition matches with [13] who showed that this is the optimal number of pebbles needed. Regarding the number of states, [13] relies on the composition of deterministic two-way transducers, which, as mentioned before, incurs at least a doubly

exponential blowup. As can be seen from (ii) above, reversibility gives a much better complexity. The high level description for composition in [4] does not give details on the number of states, and incurs n extra pebbles. As such, we do not compare with them.

4. **Uniformization of non-deterministic Reversible Pebble Transducers**. Given a non-deterministic pebble transducer $\mathcal{T}$ with k pebbles and n states, we construct a reversible pebble transducer $\mathcal{T}'$ with $2^{\mathcal{O}((kn)^2)}$ states which uniformizes the relation $[\![\mathcal{T}]\!]$ computed by $\mathcal{T}$, i.e., computes a function $[\![\mathcal{T}']\!] \subseteq [\![\mathcal{T}]\!]$ with same domain.

   The uniformization [13] of non-deterministic to deterministic pebble transducers is done using results from [14], where uniformization is done via a sequence of tranformations, namely: (*i*) transductions computed by a 0-pebble transducer are definable using non-determinsitic MSO transductions, which can be uniformized using deterministic MSO transductions, (*ii*) deterministic MSO transductions are computed by deterministic two-way transducers. The uniformization for a given k-pebble transducer [13] is then computed as a composition of a k-counting transduction and the deterministic two-way transducer obtained in (*ii*). Due to (*ii*), this may incur a non-elementary complexity, while our uniformization technique only has an exponential complexity.

5. **Polyregular functions as Reversible Pebble Transducers**. Polyregular functions can be realized as (a) deterministic two-way pebble transducers [4] or, (b) as a composition of sequential functions, squaring and iterated reverse functions [5]. This results in two possibilities to compute a reversible pebble transducer for a polyregular function.

   a. If the polyregular function is presented as a deterministic two-way pebble transducers then we may use the uniformization construction of Item 4 to obtain an equivalent reversible pebble transducer.

   b. If a polyregular function f is given as a modular expression involving sequential, squaring and iterated reverse functions, then one can directly synthesize a reversible pebble transducer computing f using (*i*) the reversible transducers for squaring and iterated reverse given in Figures 1 and 5 (*ii*) reversible transducers for the sequential functions obtained using [10, Theorem 2], and (*iii*) our composition of the resultant reversible pebble transducers. Note that the reversible machines in Figures 1 and 5 are small with 6/5 states and 1/0 pebbles respectively. Once again, this is better than constructing first from the modular expression a deterministic pebble transducer for function f and then uniformizing it to get a reversible one.

2 Reversible Pebble Transducers

Pebble Transducers with Equality Tests

In this section, we consider pebble transducers enhanced with equality tests between pebble positions, in addition to the classical query checking only if a pebble is present/absent at the head position. The equality tests are inspired from [5], where $\leq$-comparisons between any two pebble positions, as well as between the head and any pebble are allowed. Enhancing the basic tests, on whether a pebble is present/absent at the head, by allowing to check for any pair of pebbles whether or not they are at the same position is useful, especially when composing two pebble transducers.

A k-pebble automaton with equality tests (k-$\mathsf{PA}_=$) is given by a tuple $\mathcal{A} = (Q, \Sigma, \delta, k, q_i, q_f)$ where Q is a finite set of states, Σ is the finite input alphabet, δ is the transition relation (see below), and $q_i, q_f \in Q$ respectively are initial and final states.

Given an input word $u \in \Sigma^*$, we define *positions* $\mathsf{pos}(u) = \{1, \dots, |u|\}$. We visualise the input of a PA as a circular word $\#u$ where $\# \notin \Sigma$ is an endmarker reached when moving right (resp. left) from the last (resp. first) position of u. Keeping this in mind, we work with an *extended* set of positions $\mathsf{epos}(u) = \{0, 1, \dots, |u|\}$. For instance, $\begin{smallmatrix} \# & a & b & c & d \\ 0 & 1 & 2 & 3 & 4 \end{smallmatrix}$.

The head of the automaton during a run over u, occupies a position from $\mathsf{epos}(u)$, denoted h. The states Q of the pebble automaton are partitioned into 3 disjoint sets, $Q_{+1} \uplus Q_0 \uplus Q_{-1}$ signifying movement of the head to the right, no movement of the head and movement of the head to the left, respectively. Given $\mathsf{h} \in \mathsf{epos}(u)$ and a state $q \in Q$, we define $\mathsf{h} + q = \mathsf{h}$ if $q \in Q_0$; $\mathsf{h} + q = (\mathsf{h} + 1 \mod (|u| + 1))$ if $q \in Q_{+1}$; and $\mathsf{h} + q = (\mathsf{h} - 1 \mod (|u| + 1))$ if $q \in Q_{-1}$. The intuition is that if the head moves to the right of the last position of the word, then it reads #, and if the head moves to the left from #, it reads the last position of u.

Likewise, the pebbles are dropped on $\mathsf{epos}(u)$ and follow a stack policy. We denote by $\mathsf{peb} \in \mathsf{epos}(u)^{\leq k}$ the stack of positions where pebbles are dropped.

A configuration of a k-pebble automaton on the input word u is a tuple $C = (q, \mathsf{peb}, \mathsf{h}) \in Q \times \mathsf{epos}(u)^{\leq k} \times \mathsf{epos}(u)$. In this configuration, the head of the automaton reads position h, there are $|\mathsf{peb}|$ pebbles dropped and the i-th pebble ($1 \leq i \leq |\mathsf{peb}|$) is dropped on position $\mathsf{peb}_i \in \mathsf{epos}(u)$. The initial configuration is $(q_i, \varepsilon, 0)$ and the accepting (final) configuration is $(q_f, \varepsilon, 0)$. Hence, in order to accept, the automaton should lift all pebbles and move back to the endmarker. Wlog, we assume that $q_i, q_f \in Q_0$ with $q_i \neq q_f$, and there are no transitions starting from q_f or going to q_i.

Transitions are of the form $t = (q, a, \varphi, \mathsf{op}, q')$ with states $q, q' \in Q$, input letter $a \in \Sigma_\# = \Sigma \cup \{\#\}$, test φ which is a conjunction of atoms of the form $(\mathsf{h} = \mathsf{p}_i)$, $\neg(\mathsf{h} = \mathsf{p}_i)$, $(\mathsf{p}_i = \mathsf{p}_j)$ or $\neg(\mathsf{p}_i = \mathsf{p}_j)$ (with $1 \leq i, j \leq k$), and operation op of the form[1] drop_i, lift_i or nop (with $1 \leq i \leq k$). Transition t is *enabled* at configuration $C = (q, \mathsf{peb}, \mathsf{h})$ (denoted $C \models t$) if:

1. the letter read is a, i.e., $a = (\#u)_{\mathsf{h}}$
2. $\mathsf{peb}, \mathsf{h} \models \varphi$: the test succeeds, where
   - $\mathsf{peb}, \mathsf{h} \models (\mathsf{h} = \mathsf{p}_i)$ if $1 \leq i \leq |\mathsf{peb}|$ and $\mathsf{peb}_i = \mathsf{h}$,
   - $\mathsf{peb}, \mathsf{h} \models (\mathsf{p}_i = \mathsf{p}_j)$ if $1 \leq i, j \leq |\mathsf{peb}|$ and $\mathsf{peb}_i = \mathsf{peb}_j$,
3. $\mathsf{peb}, \mathsf{h} \models \mathsf{op}$: the operation can be executed where
   - $\mathsf{peb}, \mathsf{h} \models \mathsf{nop}$ is always true, and we let $\mathsf{nop}(\mathsf{peb}, \mathsf{h}) = \mathsf{peb}$,
   - $\mathsf{peb}, \mathsf{h} \models \mathsf{drop}_i$ if $|\mathsf{peb}| = i - 1$, and we let $\mathsf{drop}_i(\mathsf{peb}, \mathsf{h}) = \mathsf{peb} \cdot \mathsf{h}$,
   - $\mathsf{peb}, \mathsf{h} \models \mathsf{lift}_i$ if $|\mathsf{peb}| = i$ and $\mathsf{h} = \mathsf{peb}_i$, and we let $\mathsf{lift}_i(\mathsf{peb}, \mathsf{h}) = \mathsf{peb}_1 \cdots \mathsf{peb}_{i-1}$.

When $C \models t$ we have $C \xrightarrow{t} C' = (q', \mathsf{peb}', \mathsf{h}')$ where $\mathsf{peb}' = \mathsf{op}(\mathsf{peb}, \mathsf{h})$ and $\mathsf{h}' = \mathsf{h} + q'$.

Notice that the top pebble may be lifted from the pebble stack only when it is on the position being read. We often simply write p_i (resp. $\neg\mathsf{p}_i$) for the atomic test $(\mathsf{h} = \mathsf{p}_i)$ (resp. $\neg(\mathsf{h} = \mathsf{p}_i)$). Note that, if a transition $t = (q, a, \mathsf{p}_i, \mathsf{drop}_j, q')$ is enabled at some configuration C, then $i < j$ and $C' \models (\mathsf{p}_i = \mathsf{p}_j)$ where $C \xrightarrow{t} C'$.

An equality test $\mathsf{p}_i = \mathsf{p}_i$ allows to check if the size of the pebble stack is at least i. This is not possible if we only use atoms of the form p_i or $\neg\mathsf{p}_i$ (which is not an expressivity problem since one may always store the size of the pebble stack in the state of the automaton).

A run of $\mathcal{A}$ is a sequence $C_0 \xrightarrow{t_1} C_1 \cdots \xrightarrow{t_n} C_n$ where C_ℓ are configurations on the input word u and $t_\ell \in \delta$ are transitions of $\mathcal{A}$. The run is initial if $C_0 = (q_i, \varepsilon, 0)$ and final if $C_n = (q_f, \varepsilon, 0)$. It is accepting if it is both initial and final.

[1] We chose to specify with index i of drop and lift which pebble is dropped or lifted. This allows to get determinism or reverse-determinism without adding extra tests to transitions. We could have simply used drop and lift at the expense of adding tests to transitions to ensure (reverse-) determinism.

The automaton is *deterministic* if for all pairs of transitions $t = (q, a, \varphi, \mathsf{op}, r)$ and $t' = (q, a, \varphi', \mathsf{op}', r')$, if t and t' may be simultaneously enabled (at some configuration C) then $t = t'$. The fact that an operation op is enabled may be written as a test $\overline{\mathsf{op}}$: $\overline{\mathsf{nop}} = \mathsf{true}$, $\overline{\mathsf{drop}_i} = (\mathsf{p}_{i-1} = \mathsf{p}_{i-1}) \wedge \neg(\mathsf{p}_i = \mathsf{p}_i)$ (with $\mathsf{p}_0 = \mathsf{p}_0$ identified with true), and $\overline{\mathsf{lift}_i} = (\mathsf{h} = \mathsf{p}_i) \wedge \neg(\mathsf{p}_{i+1} = \mathsf{p}_{i+1})$. Then, the transitions t and t' may be simultaneously enabled if and only if the test $\varphi \wedge \overline{\mathsf{op}} \wedge \varphi' \wedge \overline{\mathsf{op}'}$ is satisfiable. This gives a more syntactic definition of determinism. In particular, a transtion t with operation lift_i and a transition t' with atomic test $\neg\mathsf{p}_i$ cannot be simultaneously enabled.

We define the reverse op^r of an operation op by $\mathsf{nop}^r = \mathsf{nop}$, $\mathsf{drop}_i^r = \mathsf{lift}_i$ and $\mathsf{lift}_i^r = \mathsf{drop}_i$. Notice that if $\mathsf{peb}, \mathsf{h} \models \mathsf{op}$ then $\mathsf{op}(\mathsf{peb}, \mathsf{h}), \mathsf{h} \models \mathsf{op}^r$ and $\mathsf{peb} = \mathsf{op}^r(\mathsf{op}(\mathsf{peb}, \mathsf{h}), \mathsf{h})$.

A transition $t = (q, a, \varphi, \mathsf{op}, q')$ is *reverse-enabled* at configuration $C' = (q', \mathsf{peb}', \mathsf{h}')$ (denoted $C' \models_{\mathsf{rev}} t$) if there is a configuration $C = (q, \mathsf{peb}, \mathsf{h})$ enabling t such that $C \xrightarrow{t} C'$. More explicitly, the only possibility for configuration C is given by $\mathsf{h} = \mathsf{h}' - q'$ (corresponding to $\mathsf{h}' = \mathsf{h} + q'$), and $\mathsf{peb} = \mathsf{op}^r(\mathsf{peb}', \mathsf{h})$ (corresponding to $\mathsf{peb}' = \mathsf{op}(\mathsf{peb}, \mathsf{h})$). Then, $C' \models_{\mathsf{rev}} t$ iff $\mathsf{peb}', \mathsf{h}' - q' \models \mathsf{op}^r$ and $\mathsf{op}^r(\mathsf{peb}', \mathsf{h}' - q'), \mathsf{h}' - q' \models \varphi$. In this case, we write $C = t^{-1}(C') = (q, \mathsf{op}^r(\mathsf{peb}', \mathsf{h}' - q'), \mathsf{h}' - q')$. In particular, a transtion t with operation drop_i and a transition t' with atomic test $\neg\mathsf{p}_i$ and operation nop cannot be simultaneously reverse-enabled.

The automaton is *reverse-deterministic* if for all pairs of transitions $t = (q, a, \varphi, \mathsf{op}, r)$ and $t' = (q', a, \varphi', \mathsf{op}', r)$, if t and t' may be simultaneously reverse-enabled (at some configuration C') then $t = t'$.

The pebble automaton is *reversible* if it is both *deterministic* and *reverse-deterministic*.

A k-pebble transducer with equality tests (k-$\mathsf{PT}_=$) $\mathcal{T} = (Q, \Sigma, \delta, k, q_i, q_f, \Gamma, \mu)$ is a k-pebble automaton (k-$\mathsf{PA}_=$) $\mathcal{A} = (Q, \Sigma, \delta, k, q_i, q_f)$ extended with an output alphabet Γ and an output function μ mapping each transition of $\mathcal{A}$ to a word in Γ^*. The semantics of $\mathcal{T}$ is a relation $[\![\mathcal{T}]\!] \subseteq \Sigma^* \times \Gamma^*$ consisting of all pairs (u, v) such that there is an accepting run $C_0 \xrightarrow{t_1} C_1 \cdots \xrightarrow{t_n} C_n$ on $\#u$ with $v = \mu(t_1) \cdots \mu(t_n)$.

A pebble transducer is deterministic (reverse-deterministic, reversible) whenever the underlying pebble automaton is. The semantics of a deterministic (reverse-deterministic, reversible) pebble transducer is a partial function.

▶ Remark 2.1. Even though the semantics of 0-pebble transducers is different to that of two-way transducers as defined in [10], the two models are equivalent. Given a transducer with n states using one semantics, one can build a transducer realising the same relation using the other semantics, with $\mathcal{O}(n)$ states. Moreover, determinism and reverse-determinism are preserved by this translation (see the extended version [11]).

Basic Pebble Transducers

A (basic) pebble automaton (k-PA) or transducer (k-PT) is one in which equality tests of the form $(\mathsf{p}_i = \mathsf{p}_j)$ or $\neg(\mathsf{p}_i = \mathsf{p}_j)$ are not used. In some cases, we use a bit vector $\overline{b} = (b_1, \ldots, b_k) \in \{0, 1\}^k$ to denote a full test φ which is a conjunction over all $1 \leq i \leq k$ of atoms p_i if $b_i = 1$ and $\neg\mathsf{p}_i$ if $b_i = 0$.

▶ **Example 2.2.** Figure 1 depicts a reversible 1-pebble transducer computing the squaring function. This function outputs the concatenation of as many copies of the input word as there are letters in it. For the i^{th} copy, the i^{th} letter is marked. The blue part of the transducer copies its input and marks the positions on which the unique pebble is dropped. The black and red part initially drops the pebble on the first position, then lifts the pebble and drops it on the next position at each iteration. It moves to state q_2 and ends when the pebble is lifted from the last position. Without looking at the pebble operations, the transducer is

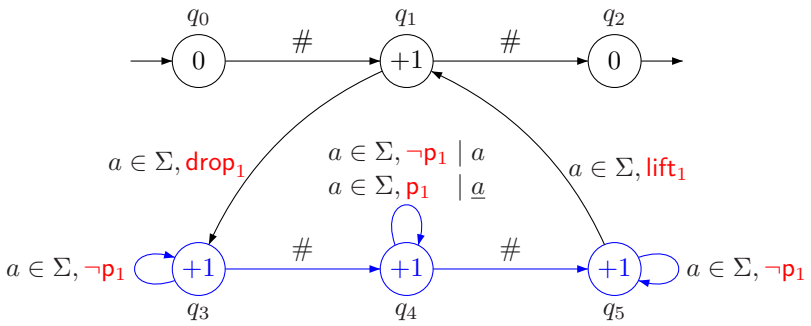


Figure 1 Reversible 1-pebble transducer for the function *squaring*. The labels $i \in \{0, +1, -1\}$ inside the states represent that they lie in Q_i. To keep the figure light, ε-outputs of transitions are omitted. The same transducer with state q_3 in Q_{-1} would also compute the squaring function.

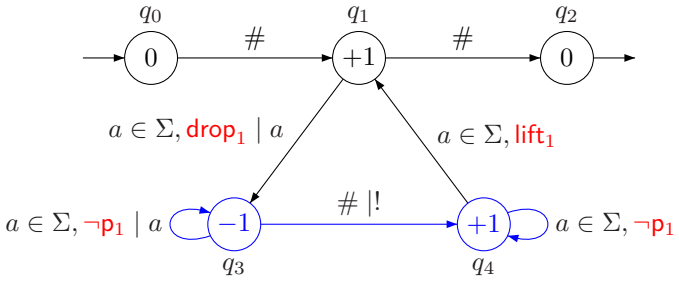


Figure 2 Reversible 1-pebble transducer for the function *all prefixes in reverse*.

deterministic (resp. reverse-deterministic) at every state except state q_5 (resp. q_3). However, since the lift_1 operation can only be triggered at the position on which the pebble 1 is dropped, the pebble guards lift_1 and $\neg\mathsf{p}_1$ cannot be simultaneously enabled, and thus the transducer is deterministic at q_5. Symmetrically, the reverse of a drop_1 operation can only be enabled on the position marked by p_1, and thus drop_1 and $\neg\mathsf{p}_1$ cannot be simultaneously reverse-enabled, and then the transducer is reverse-deterministic at q_3, proving that we have a reversible transducer.

▶ **Example 2.3.** Figure 2 represents a reversible 1-pebble transducer for the function which maps a word to the concatenation of the reverse of all its prefixes, with a ! symbol between each one. A run of this transducer on the word *abb* produces the word *a*!*ba*!*bba*!.
Using the same arguments as in the previous example, one can show that the transducer is reversible.

3 Simulating equality tests of pebbles

In this section, we show the equivalence between a basic k pebble transducer (k-PT) and a k pebble transducer allowing equality tests (k-PT$_=$). The precise statement is given by Theorem 3.1, followed by a proof sketch, the full proof can be found in [11].

▶ **Theorem 3.1.** *Given a k-pebble transducer with equality tests $\mathcal{A}$ having n states, one can construct a basic k-pebble transducer $\mathcal{B}$ with $n2^{k^2}$ states such that $[\![\mathcal{A}]\!] = [\![\mathcal{B}]\!]$.*

Moreover, if $\mathcal{A}$ is deterministic (reverse-deterministic, reversible) then so is $\mathcal{B}$.

Sketch of proof. The main idea while simulating $\mathcal{A}$ is to keep track of which pairs of pebbles are on the same position in a coherent manner: that is, if $p_i = p_j$ and $p_j = p_\ell$ then we also have $p_i = p_\ell$. While constructing $\mathcal{B}$, we store the equalities of the positions of the k pebbles as a $k \times k$ boolean matrix M where $M_{i,i} = 1$ represents that the i^{th} pebble has been dropped, and $M_{i,j} = 1$ if the i^{th} and j^{th} pebbles are on the same position. Then if we denote by Q the set of states of $\mathcal{A}$, the set of states of $\mathcal{B}$ is $Q' = Q \times \{0,1\}^{k^2}$.

We now explain how the information is used and updated. Initially, no pebble is dropped, and the initial state of $\mathcal{B}$ is $(q_i, \overline{0})$ where $\overline{0}$ denotes the $k \times k$ zero matrix and q_i is the initial state of $\mathcal{A}$. Similarly, a run is accepting only when all pebbles are lifted. Thus the final state of $\mathcal{B}$ is $(q_f, \overline{0})$ where q_f is the final state of $\mathcal{A}$. Recall that basic pebble transducers can see whether a given pebble is at the same position as the reading head, thus transitions of $\mathcal{A}$ involving a guard of the form $\mathsf{h} = \mathsf{p}_i$ or $\neg(\mathsf{h} = \mathsf{p}_i)$ can be kept in $\mathcal{B}$. Tests of the form $\mathsf{p}_i = \mathsf{p}_j$ are replaced by true if $M_{i,j} = 1$ and by false otherwise. On transitions where there is no pebble lifting or dropping, the matrix M does not need to be updated. The i^{th} pebble p_i can be dropped only if the $(i-1)^{th}$ pebble is already dropped (that is, $M_{i-1,i-1} = 1$) and the i^{th} one is not ($M_{i,i} = 0$). On dropping p_i, the matrix M is updated by setting all coefficients $M_{i,j}$ and $M_{j,i}$ to 1 if $j = i$ or ($j < i$ and $\mathsf{h} = p_j$), and 0 otherwise. Note that to do this, each transition of $\mathcal{A}$ with a drop_i action is duplicated into 2^{i-1} disjoint transitions testing which subsets of $\mathsf{p}_1, \ldots, \mathsf{p}_{i-1}$ are present at the head. Only one transition can then be fired, depending on the bit vector generated by the pebbles at the current position. Similarly, the i^{th} pebble p_i can be lifted if and only if $\mathsf{h} = \mathsf{p}_i$ and no larger pebbles are dropped ($i = k$ or $M_{i+1,i+1} = 0$). On lifting p_i, the matrix M is updated by setting all coefficients of the i^{th} row and column to 0. However to ensure the preservation of reverse-determinism, each lift_i transition is duplicated into 2^{i-1} transitions that have complete and disjoint tests for the current bit vector over pebbles $\mathsf{p}_1, \ldots, \mathsf{p}_{i-1}$.

Finally, suppose that $\mathcal{A}$ is reversible. The only extra transitions added to $\mathcal{B}$ are when a pebble is dropped or lifted. However, all the duplicate transitions that have been added in $\mathcal{B}$ have disjoint tests (enforced by disjoint bit vectors). Thus $\mathcal{B}$ is deterministic since $\mathcal{A}$ is. On the other hand, consider we want to reverse a lift_i transition in $\mathcal{A}$. This means that the i^{th} pebble was at the previous position of the reading head, and further, there is only one bit vector (enriching the ith row and column of M) which was valid at the previous step. The correct predecessor matrix is then the only one where the i^{th} row and column is coherent with the bit vector of the previous position. ◀

4 Composition of pebble transducers

The goal of this section is to give an efficient (in the size of the resulting transducer) construction for the composition of pebble transducers. Since the construction is rather involved, we start with two simpler cases. The first one described in Section 4.1 shows how to compose a reversible n-pebble transducer with the reverse function $\mathsf{rev}\colon \Gamma^* \to \Gamma^*$ which takes a string of letters and outputs the sequence of letters in reverse order. For instance, $\mathsf{rev}(abac) = caba$. The second simpler case is the composition of a reversible n-pebble transducer with a deterministic 0-pebble transducer. This case is addressed in Section 4.2. Finally, the general case is solved in Section 4.3.

4.1 Reversing the output of a reversible pebble transducer

Let $\mathcal{T} = (Q, \Sigma, \delta, n, q_i, q_f, \Gamma, \mu)$ be a reversible n-pebble transducer, possibly with equality tests. The goal is to construct a reversible transducer $\mathcal{T}^r = (Q', \Sigma, \delta', n, q'_i, q'_f, \Gamma, \mu')$ which outputs the reverse of the string produced by $\mathcal{T}$: $\mathsf{dom}(\mathcal{T}^r) = \mathsf{dom}(\mathcal{T})$ and for all $u \in \mathsf{dom}(\mathcal{T})$, the string $[\![\mathcal{T}^r]\!](u)$ is the reverse of the string $[\![\mathcal{T}]\!](u)$.

The set of states of $\mathcal{T}^r$ is $Q' = Q$ but the polarity of the states is reversed: $Q'_{+1} = Q_{-1}$, $Q'_0 = Q_0$ and $Q'_{-1} = Q_{+1}$. The initial and final states are exchanged: $q'_i = q_f$ and $q'_f = q_i$. The transitions are also reversed in the following way. Let $t = (q, a, \varphi, \mathsf{op}, q')$ be a transition of $\mathcal{T}$. Then, $t^r = (q', a, (\varphi, \mathsf{op})^r, q)$ is a transition of $\mathcal{T}^r$ with $\mu'(t^r) = \mu(t)$. We define $(\varphi, \mathsf{op})^r = \mathsf{op}(\varphi), \mathsf{op}^r$: the operation is simply op^r but the test $\mathsf{op}(\varphi)$ depends on both φ and op. It remains to define $\mathsf{op}(\varphi)$. We want that $\mathsf{peb}, \mathsf{h} \models \varphi$ iff $\mathsf{op}(\mathsf{peb}, \mathsf{h}), \mathsf{h} \models \mathsf{op}(\varphi)$, assuming that $\mathsf{op}(\mathsf{peb}, \mathsf{h})$ is defined. We let $\mathsf{op}(\varphi_1 \wedge \varphi_2) = \mathsf{op}(\varphi_1) \wedge \mathsf{op}(\varphi_2)$ and $\mathsf{op}(\neg\varphi) = \neg\mathsf{op}(\varphi)$. We let $\mathsf{nop}(\varphi) = \varphi$, and the remaining cases are given below

$$\mathsf{drop}_\ell(\mathsf{h} = \mathsf{p}_i) = \begin{cases} (\mathsf{h} = \mathsf{p}_i) & \text{if } i < \ell \\ \mathsf{false} & \text{otherwise} \end{cases} \qquad \mathsf{drop}_\ell(\mathsf{p}_i = \mathsf{p}_j) = \begin{cases} (\mathsf{p}_i = \mathsf{p}_j) & \text{if } i, j < \ell \\ \mathsf{false} & \text{otherwise} \end{cases}$$

$$\mathsf{lift}_\ell(\mathsf{h} = \mathsf{p}_i) = \begin{cases} (\mathsf{h} = \mathsf{p}_i) & \text{if } i < \ell \\ \mathsf{true} & \text{if } i = \ell \\ \mathsf{false} & \text{otherwise} \end{cases} \qquad \mathsf{lift}_\ell(\mathsf{p}_i = \mathsf{p}_j) = \begin{cases} (\mathsf{p}_i = \mathsf{p}_j) & \text{if } i, j < \ell \\ (\mathsf{h} = \mathsf{p}_i) & \text{if } i < j = \ell \\ (\mathsf{h} = \mathsf{p}_j) & \text{if } j < i = \ell \\ \mathsf{true} & \text{if } i = j = \ell \\ \mathsf{false} & \text{otherwise} \end{cases}$$

We can easily check that $\mathsf{peb}, \mathsf{h} \models \varphi$ iff $\mathsf{op}(\mathsf{peb}, \mathsf{h}), \mathsf{h} \models \mathsf{op}(\varphi)$ when $\mathsf{op}(\mathsf{peb}, \mathsf{h})$ is defined. Recall that, when $\mathsf{peb}' = \mathsf{op}(\mathsf{peb}, \mathsf{h})$ is defined, then $\mathsf{op}^r(\mathsf{peb}', \mathsf{h})$ is defined and we have $\mathsf{peb} = \mathsf{op}^r(\mathsf{peb}', \mathsf{h})$. With $t = (q, a, \varphi, \mathsf{op}, q')$ and $t^r = (q', a, \mathsf{op}(\varphi), \mathsf{op}^r, q)$, we deduce that

$$(q, \mathsf{peb}, \mathsf{h}) \xrightarrow{t} (q', \mathsf{peb}', \mathsf{h}') \quad \text{iff} \quad (q', \mathsf{peb}', \mathsf{h}' - q') \xrightarrow{t^r} (q, \mathsf{peb}, \mathsf{h} - q) \tag{1}$$

We deduce that, for $u \in \Sigma^*$, the following sequence is a run of $\mathcal{T}$ on $\#u$

$$(q_1, \mathsf{peb}^1, \mathsf{h}_1) \xrightarrow{t_1} (q_2, \mathsf{peb}^2, \mathsf{h}_2) \xrightarrow{t_2} (q_3, \mathsf{peb}^3, \mathsf{h}_3) \cdots \xrightarrow{t_{m-1}} (q_m, \mathsf{peb}^m, \mathsf{h}_m)$$

if and only if the following sequence is a run of $\mathcal{T}^r$ on $\#u$

$$(q_1, \mathsf{peb}^1, \mathsf{h}_1 - q_1) \xleftarrow{t_1^r} (q_2, \mathsf{peb}^2, \mathsf{h}_2 - q_2) \xleftarrow{t_2^r} (q_3, \mathsf{peb}^3, \mathsf{h}_3 - q_3) \cdots \xleftarrow{t_{m-1}^r} (q_m, \mathsf{peb}^m, \mathsf{h}_m - q_m)$$

Now, the run of $\mathcal{T}$ is accepting iff $(q_1, \mathsf{peb}^1, \mathsf{h}_1) = (q_i, \varepsilon, 0)$ and $(q_m, \mathsf{peb}^m, \mathsf{h}_m) = (q_f, \varepsilon, 0)$. Since $q_i, q_f \in Q_0$, we deduce that the run of $\mathcal{T}$ is accepting if and only if the corresponding run of $\mathcal{T}^r$ is accepting, i.e., $(q_m, \mathsf{peb}^m, \mathsf{h}_m - q_m) = (q'_i, \varepsilon, 0) = (q_f, \varepsilon, 0)$ and $(q_1, \mathsf{peb}^1, \mathsf{h}_1 - q_1) = (q'_f, \varepsilon, 0) = (q_i, \varepsilon, 0)$.

We deduce that $\mathsf{dom}(\mathcal{T}^r) = \mathsf{dom}(\mathcal{T})$ and, by definition of μ', for $u \in \mathsf{dom}(\mathcal{T})$, the string $[\![\mathcal{T}^r]\!](u)$ is the reverse of the string $[\![\mathcal{T}]\!](u)$.

To show that $\mathcal{T}^r$ is deterministic, consider a pair of transitions $t_1^r = (q', a, (\varphi_1, \mathsf{op}_1)^r, q_1)$ and $t_2^r = (q', a, (\varphi_2, \mathsf{op}_2)^r, q_2)$ where $t_1 = (q_1, a, \varphi_1, \mathsf{op}_1, q')$ and $t_2 = (q_2, a, \varphi_2, \mathsf{op}_2, q')$ are transitions of $\mathcal{T}$. Assume that t_1^r is enabled at some configuration $C' = (q', \mathsf{peb}', \mathsf{h}')$. Using (1), we deduce that t_1 is reverse-enabled at configuration $C = (q', \mathsf{peb}', \mathsf{h}' + q')$. Similarly, if t_2^r is enabled at C' then t_2 is reverse-enabled at C. Since $\mathcal{T}$ is reverse-deterministic, we deduce that $\mathcal{T}^r$ is deterministic.

We can prove similarly that $\mathcal{T}^r$ is reverse-deterministic using the fact that $\mathcal{T}$ is determinisitic. In particular, we show using (1) that transition t^r is reversed-enabled at some configuration $C' = (q, \mathsf{peb}, \mathsf{h})$ if and only if transition t is enabled at configuration $C = (q, \mathsf{peb}, \mathsf{h} + q)$. We deduce that $\mathcal{T}^r$ is reversible.

4.2 Composition: Simple case

In this subsection, we prove the following result.

▶ **Theorem 4.1.** *Let* $\mathcal{T} = (Q, \Sigma, \delta, n, q_i, q_f, \Gamma, \mu)$ *be a* reversible *n-pebble transducer (possibly with equality tests) computing a function* $f\colon \Sigma^* \to \Gamma^*$. *Let* $\mathcal{T}' = (Q', \Gamma, \delta', 0, q'_i, q'_f, \Delta, \mu')$ *be a* deterministic *0-pebble transducer computing a function* $g\colon \Gamma^* \to \Delta^*$. *We can construct a* deterministic *n-pebble transducer $\mathcal{T}''$ with at most* $2 \cdot |Q| \cdot [Q'|$ *many states computing the composition* $g \circ f\colon \Sigma^* \to \Delta^*$. *Moreover, if $\mathcal{T}'$ is* reversible *then so is $\mathcal{T}''$.*

Proof. Wlog, we assume that transducer $\mathcal{T}$ outputs at most one letter on each transition, i.e., $\mu(t) \in \Gamma \cup \{\varepsilon\}$ for each transition t of $\mathcal{T}$. This can be done while preserving reversibility in the following way: a transition $t = (q, a, \varphi, \mathsf{op}, s)$ producing $v_1 \cdots v_n$ $(n \geq 2)$ is decomposed into n transitions $t_1 = (q, a, \varphi \wedge \overline{\mathsf{op}}, \mathsf{nop}, (q, v_1))$, $t_n = ((q, v_1 \dots v_{n-1}), a, \varphi, \mathsf{op}, s)$ and for $1 < i < n$, $t_i = ((q, v_1 \dots v_{i-1}), a, \varphi \wedge \overline{\mathsf{op}}, \mathsf{nop}, (q, v_1 \dots v_i))$. We set $\mu(t_i) = v_i$. Since we maintain the tests φ and $\overline{\mathsf{op}}$ in the sequence of transitions t_i, even if a state (q, v) happens to be used by several transitions they should be disjoint due to the reversibility of the inital transducer. It will be also more convenient to assume that, when running on $\#u$ with $u \in \Sigma^*$, the transducer $\mathcal{T}$ writes $\#v$ with $v \in \Gamma^*$. This is achieved by setting $\mu(t_0) = \#$ where $t_0 = (q_i, \#, \mathsf{true}, -, -)$ is the unique transition enabled at the initial configuration $C_0 = (q_i, \varepsilon, 0)$ of $\mathcal{T}$ (pebble stack empty and head on position 0). Finally, we also assume that the transducer $\mathcal{T}'$ always fully read its input. This will allow us to ensure that the transducer $\mathcal{T}''$ only accepts words that belong to $\mathsf{dom}(\mathcal{T})$. This can be achieved by addding to $\mathcal{T}'$ a single state $r \in Q'_{+1}$, and if $(q'_i, \#, \varphi, \mathsf{op}, s)$ is the initial transition of δ', we replace it with $(q'_i, \#, \mathsf{true}, \mathsf{nop}, r)$, $(r, a, \mathsf{true}, \mathsf{nop}, r)$ for $a \neq \#$ and $(r, \#, \varphi, \mathsf{op}, s)$. This construction preserves reversibility.

We construct the n-pebble transducer $\mathcal{T}'' = (Q'', \Sigma, \delta'', n, q''_i, q''_f, \Delta, \mu'')$ as a synchronized product of $\mathcal{T}$ and $\mathcal{T}'$. The set of states is $Q'' = (Q \times Q') \cup (\widehat{Q} \times (Q'_{-1} \cup Q'_{+1}))$ where $\widehat{Q} = \{\hat{q} \mid q \in Q\}$ is a disjoint copy of Q. We have $|Q''| \leq 2 \cdot |Q| \cdot [Q'|$. We define

$$
\begin{aligned}
Q''_0 &= (Q \times Q') \cup (\widehat{Q_0} \times (Q'_{-1} \cup Q'_{+1})) \\
Q''_{+1} &= (\widehat{Q_{+1}} \times Q'_{+1}) \cup (\widehat{Q_{-1}} \times Q'_{-1}) \\
Q''_{-1} &= (\widehat{Q_{+1}} \times Q'_{-1}) \cup (\widehat{Q_{-1}} \times Q'_{+1})\,.
\end{aligned}
$$

The initial state is $q''_i = (q_i, q'_i) \in Q''_0$ and the final state is $q''_f = (q_i, q'_f) \in Q''_0$.

The intuition is that, in a state $(q, q') \in Q \times Q'$, transducer $\mathcal{T}''$ will synchronize a pair (t, t') of transitions of $\mathcal{T}$ and $\mathcal{T}'$ where the output $\mu(t) \in \Gamma$ of t is the input letter of t'. On the other hand, when $\mathcal{T}''$ is in a state $(\hat{q}, q')$, it will simulate the computation of $\mathcal{T}$ forward if $q' \in Q'_{+1}$ (resp. backward if $q' \in Q'_{-1}$) using transitions t of $\mathcal{T}$ producing $\mu(t) = \varepsilon$ until the computation reaches a transition t of $\mathcal{T}$ producing a letter $\mu(t) \in \Gamma$.

To handle the fact that the input of $\mathcal{T}$ is a circular word, we extend δ with the transition $t_{f,i} = (q_f, \#, \neg(\mathsf{p}_1 = \mathsf{p}_1), \mathsf{nop}, q_i)$ which is only enabled in the final configuration $(q_f, \varepsilon, 0)$ of $\mathcal{T}$ and moves to the initial configuration $(q_i, \varepsilon, 0)$. We let $\mu(t_{f,i}) = \varepsilon$.

We define now the transitions δ'' and the output function μ''. Let $t = (q, a, \varphi, \mathsf{op}, s)$ be a transition of $\mathcal{T}$. If $\mu(t) = a' \in \Gamma$, for each transition $t' = (q', a', s')$ of $\mathcal{T}'$, we introduce in $\mathcal{T}''$ the synchronized transition t'' defined in Equations (tr-a)–(tr-c) with $\mu''(t'') = \mu'(t')$.

Firstly, we synchronize a transition and switch to the simulation mode that will search for either the next or previous production, depending on the polarity of the destination state. If $\mathcal{T}'$ is going to the right, we advance in the run of $\mathcal{T}$ and apply the transition t. If $\mathcal{T}'$ is going

to the left, we need to start rewinding $\mathcal{T}$ to compute the transition which led to state q:

$$t'' = (q, q') \xrightarrow{a,\varphi,\mathsf{op}} (\hat{s}, s') \qquad \text{if } s' \in Q'_{+1} \qquad \text{(tr-a)}$$

$$t'' = (q, q') \xrightarrow{a,\varphi\wedge\overline{\mathsf{op}},\mathsf{nop}} (\hat{q}, s') \qquad \text{if } s' \in Q'_{-1} \qquad \text{(tr-b)}$$

$$t'' = (q, q') \xrightarrow{a,\varphi\wedge\overline{\mathsf{op}},\mathsf{nop}} (q, s') \qquad \text{if } s' \in Q'_{0} \qquad \text{(tr-c)}$$

Secondly, if we are in simulation mode of $\mathcal{T}$, we need to keep simulating until we reach a non empty production of $\mathcal{T}$. Hence if $\mu(t) = \varepsilon$ then we stay in the simulation mode of $\mathcal{T}$. For each $q' \in Q'_{-1} \cup Q'_{+1}$, we introduce in $\mathcal{T}''$ the following transitions t'' with $\mu''(t'') = \varepsilon$. Note that depending on the polarity of q', we either need to advance or rewind the computation of $\mathcal{T}$. Rewinding is done following the constructions detailled in Section 4.1.

$$t'' = (\hat{q}, q') \xrightarrow{a,\varphi,\mathsf{op}} (\hat{s}, q') \qquad \text{if } q' \in Q'_{+1} \qquad \text{(mv-a)}$$

$$t'' = (\hat{s}, q') \xrightarrow{a,(\varphi,\mathsf{op})^r} (\hat{q}, q') \qquad \text{if } q' \in Q'_{-1} \qquad \text{(mv-b)}$$

Finally, if $\mu(t) = a' \in \Gamma$ we found the transition of $\mathcal{T}$ which produces the input letter a' to be read by $\mathcal{T}'$. Hence, we switch from the simulation mode of $\mathcal{T}$ to its synchronization mode. For each $q' \in Q_{-1} \cup Q_{+1}$, we add the following transitions t'' with $\mu''(t'') = \varepsilon$.

$$t'' = (\hat{q}, q') \xrightarrow{a,\varphi\wedge\overline{\mathsf{op}},\mathsf{nop}} (q, q') \qquad \text{if } q' \in Q'_{+1} \qquad \text{(sw-a)}$$

$$t'' = (\hat{s}, q') \xrightarrow{a,(\varphi,\mathsf{op})^r} (q, q') \qquad \text{if } q' \in Q'_{-1} \qquad \text{(sw-b)}$$

We prove in Appendix A.1 that the constructed transducer $\mathcal{T}''$ is deterministic (Claim A.1), and that $\mathcal{T}''$ is reverse-deterministic if $\mathcal{T}'$ is reversible (Claim A.2). We prove now the correctness of the construction, i.e., that $\mathcal{T}''$ computes the function $g \circ f$. We fix $u \in \mathsf{dom}(f) \subseteq \Sigma^*$ and $v = f(u) \in \Gamma^*$. We consider the accepting run of $\mathcal{T}$ on $\#u$ producing $\#v$:

$$C_0 = (q_i, \varepsilon, 0) \xrightarrow{t_0} C_1 \xrightarrow{t_1} C_2 \xrightarrow{t_2} \cdots C_N \xrightarrow{t_N} C_{N+1} = (q_f, \varepsilon, 0)$$

where $C_\ell = (q_\ell, \mathsf{peb}_\ell, \mathsf{h}_\ell)$ are configurations and $t_\ell \in \delta \setminus \{t_{f,i}\}$ are transitions of $\mathcal{T}$. Notice that the configurations in this accepting run are pairwise distinct. This follows from the fact that the automaton is deterministic and there are no transitions (other than $t_{f,i}$) starting from the final state q_f.

For each position $i \in \mathsf{epos}(v)$, let $\mathsf{encode}(i)$ be the index of the transition in the above run producing the letter at position i in $\#v$. Since $\mu(t_0) = \#$ we have $j_0 = 0$. Also, $\mu(t_\ell) = \varepsilon$ if $\mathsf{encode}(i) < \ell < \mathsf{encode}(i+1)$ and $\mu(t_{\mathsf{encode}(i)}) \in \Gamma \cup \{\#\}$ if $i \in \mathsf{epos}(v)$.

The main idea is to encode the head position h' of $\mathcal{T}'$ by the configuration $C_{\mathsf{encode}(\mathsf{h}')}$. More precisely, the encoding of a configuration $C' = (q', \mathsf{h}')$ of $\mathcal{T}'$ on $\#v$ is defined as $\mathsf{encode}(C') = ((q_\ell, q'), \mathsf{peb}_\ell, \mathsf{h}_\ell)$ where $\ell = \mathsf{encode}(\mathsf{h}')$. The proof of Claim 4.2 is in Appendix A.1.

▷ Claim 4.2. There is a transition $(q', \mathsf{h}') \xrightarrow{t'} (s', \mathsf{h}' + s')$ of $\mathcal{T}'$ on $\#v$ if and only if there is a nonempty run $\mathsf{encode}((q', \mathsf{h}')) \xrightarrow{+} \mathsf{encode}((s', \mathsf{h}' + s'))$ of $\mathcal{T}''$ on $\#u$ which does not use an intermediate state in $Q \times Q'$.

Now, we can show that the transducer $\mathcal{T}''$ computes the function $g \circ f$. With the notation above, assuming that $v \in \mathsf{dom}(g)$ we consider the accepting run ρ' of $\mathcal{T}'$ on $\#v$:

$$C'_0 = (q'_i, 0) \xrightarrow{t'_0} C'_1 \xrightarrow{t'_1} C'_2 \cdots C'_{N'} \xrightarrow{t'_{N'}} C'_{N'+1} = (q'_f, 0)\,. \qquad (2)$$

Using Claim 4.2 we obtain the accepting run ρ'' of $\mathcal{T}''$ on $\#u$

$$\mathsf{encode}(C'_0) = ((q_i, q'_i), \varepsilon, 0) \xrightarrow{+} \mathsf{encode}(C'_1) \xrightarrow{+} \mathsf{encode}(C'_2) \\ \cdots \mathsf{encode}(C'_{N'}) \xrightarrow{+} \mathsf{encode}(C'_{N'+1}) = ((q_i, q'_f), \varepsilon, 0)\,.$$

It is easy to see that ρ'' produces the same output string $g(v)$ as ρ'. We get $[\![\mathcal{T}'']\!](u) = g(v) = g \circ f(u)$.

Conversely, an accepting run ρ'' of $\mathcal{T}''$ on $\#u$ can be split according to its visits to states in $Q \times Q'$:

$$C''_0 = ((q_i, q'_i), \varepsilon, 0) \xrightarrow{+} C''_1 \xrightarrow{+} C''_2 \cdots C''_{N'} \xrightarrow{+} C''_{N'+1} = ((q_i, q'_f), \varepsilon, 0)\,.$$

where the C''_i are the configurations with state in $Q \times Q'$. We have $C''_0 = \mathsf{encode}(C'_0)$. By induction, and using Claim 4.2, we can easily show that, for $0 \leq i \leq N'$, there are transitions t'_i and configurations C'_{i+1} such that $C'_i \xrightarrow{t'_i} C'_{i+1}$ on $\#v$ and $C''_{i+1} = \mathsf{encode}(C'_{i+1})$. We deduce that (2) gives an accepting run ρ' of $\mathcal{T}'$ on $\#v$. Again, the output string $[\![\mathcal{T}'']\!](u)$ of ρ'' is the same as the output string $g(v)$ of ρ'. Because we assumed that $\mathcal{T}'$ reads its whole input, by emulating $\mathcal{T}'$ we know that the transducer $\mathcal{T}''$ fully simulates $\mathcal{T}$, which ensures that it accepts an input u if, and only if, it belongs to $\mathsf{dom}(\mathcal{T})$ and $\mathcal{T}(u)$ belongs to $\mathsf{dom}(\mathcal{T}')$.

This concludes the proof of Theorem 4.1. ◀

▶ **Example 4.3.** We illustrate the construction from Theorem 4.1, with $\mathcal{T}$ being a slight modification of the *squaring* function realized by the transducer from Figure 1, and $\mathcal{T}'$ is the function *iterated reverse* realized by the transducer from Figure 5. The *squaring* function is modified as such: instead of outputting marked letters $\underline{a}$ when reading the letter on which the pebble is placed, from state q_4, it outputs !. Now the output of this automaton is of a form that is expected for the function *iterated reverse*.

We show a fragment of the run of $\mathcal{T}''$ on the input $\#u = \#bcd$. On this input, $\mathcal{T}$ produces the output $\#v = \#!cdb!dbc!$, on which $\mathcal{T}'$ produces $!bdc!cdb$. The fragment of the run illustrated below starts when transducer $\mathcal{T}'$ reads the second ! of v, and needs to rewind the computation of $\mathcal{T}$ in order to process the infix cdb of u. Notice that to produce the second ! of v, $\mathcal{T}$ is producing the second copy of u, hence the pebble placed on the c in the initial configuration of the run. The fragment ends when $\mathcal{T}'$ is done reversing the current infix, and is about to find the previous ! symbol.

Transitions of the type (mv-b) are represented by $\rightsquigarrow$, those of the type (sw-b) by $\dashrightarrow$, and those of the type (tr-b) by $\rightarrow$. Notice that in this fragment of run, no transition where $\mathcal{T}'$ needs to go right on $\#v$ are represented. The run is as follow:

$$\big((q_4, q'_1), 2, 2\big) \xrightarrow[1]{c} \big((\hat{q}_4, q'_2), 2, 1\big) \xdashrightarrow[2]{b} \big((q_4, q'_2), 2, 1\big) \xrightarrow[3]{b|b} \big((\hat{q}_4, q'_2), 2, 0\big) \overset{\#}{\underset{4}{\rightsquigarrow}} \big((\hat{q}_3, q'_2), 2, 3\big) \\ \overset{d}{\underset{5}{\rightsquigarrow}} \big((\hat{q}_3, q'_2), 2, 2\big) \overset{c}{\underset{6}{\rightsquigarrow}} \big((\hat{q}_1, q'_2), \varepsilon, 1\big) \overset{b}{\underset{7}{\rightsquigarrow}} \big((\hat{q}_5, q'_2), 1, 0\big) \overset{\#}{\underset{8}{\rightsquigarrow}} \big((\hat{q}_4, q'_2), 1, 3\big) \\ \xdashrightarrow[9]{d} \big((q_4, q'_2), 1, 3\big) \xrightarrow[10]{d|d} \big((\hat{q}_4, q'_2), 1, 2\big) \xdashrightarrow[11]{c} \big((q_4, q'_2), 1, 2\big) \xrightarrow[12]{c|c} \big((\hat{q}_4, q'_2), 1, 1\big)$$

Notice how after transition 3, because $\mathcal{T}'$ still requires to move left, the computation of $\mathcal{T}$ is rewound and goes back to the first copy of u produced by $\mathcal{T}$. Transition 6 undoes the drop_1 operation, hence the ε in the following configuration, and transition 7 undoes the lift_1, effectively moving the pebble one position to the left.

4.3 Composition: General case

Now, we prove the general case of composition of pebble transducers.

▶ **Theorem 4.4.** *Let* $\mathcal{T} = (Q, \Sigma, \delta, n, q_i, q_f, \Gamma, \mu)$ *be a* reversible *n-pebble transducer (possibly with equality tests) computing a function* $f\colon \Sigma^* \to \Gamma^*$. *Let* $\mathcal{T}' = (Q', \Gamma, \delta', m, q'_i, q'_f, \Delta, \mu')$ *be a* deterministic *m-pebble transducer computing a function* $g\colon \Gamma^* \to \Delta^*$. *Let* $r = (n+1)(m+1)-1$. *We can construct a* deterministic *r-pebble transducer* $\mathcal{T}'' = (Q'', \Sigma, \delta'', r, q''_i, q''_f, \Delta, \mu'')$ with equality tests *computing the composition* $g \circ f\colon \Sigma^* \to \Delta^*$. *The number of states of* $\mathcal{T}''$ *is at most* $\mathcal{O}(|Q|^{m+2} \cdot |Q'| \cdot (n+1)^{m+3})$. *Moreover, if* $\mathcal{T}'$ *is* reversible *then so is* $\mathcal{T}''$.

To simplify the construction, we first show that we may restrict to reversible pebble transducers that do not move their head when dropping or lifting a pebble.

▶ **Lemma 4.5.** *For each reversible n-pebble transducer* $\mathcal{T} = (Q, \Sigma, \delta, n, q_i, q_f, \Gamma, \mu)$, *we can construct a reversible n-pebble transducer* $\mathcal{T}' = (Q', \Sigma, \delta', n, q'_i, q'_f, \Gamma, \mu')$ *with* $|Q'| \leq 3|Q|$, *computing the same function, and such that every transition in* $\mathcal{T}'$ *which moves the head has operation* nop.

Proof. Let $Q_{\mathsf{op}} = Q \times \{\mathsf{drop}, \mathsf{lift}\}$. We let $Q' = Q \cup Q_{\mathsf{op}}$, $q'_i = q_i$, $q'_f = q_f$, $Q'_0 = Q_0 \cup Q_{\mathsf{op}}$, $Q'_{+1} = Q_{+1}$ and $Q'_{-1} = Q_{-1}$. We define now δ' and μ'. Each transition $t = (q, a, \varphi, \mathsf{nop}, s)$ of $\mathcal{T}$ also appears in $\mathcal{T}'$, and for each transition $t = (q, a, \varphi, \mathsf{op}, s)$ of $\mathcal{T}$ where $\mathsf{op} \neq \mathsf{nop}$, we define the following two transitions of $\mathcal{T}'$:

$$t' = (q, a, \varphi, \mathsf{op}, (q, \mathsf{op}))$$
$$t'' = ((q, \mathsf{op}), \mathsf{op}(\varphi) \wedge \overline{\mathsf{op}^r}, \mathsf{nop}, s)$$

with output $\mu'(t') = \mu(t)$ and $\mu'(t'') = \varepsilon$. Note that we abuse notation as we forget the index of the action op in the state (q, op). We claim that $\mathcal{T}'$ is reversible (see [11]).

Finally, it is easy to see that there is a one-to-one correspondence between the accepting runs of $\mathcal{T}$ and the accepting runs of $\mathcal{T}'$, moreover this correspondence preserves the output string produced. Hence, $[\![\mathcal{T}]\!] = [\![\mathcal{T}']\!]$. ◀

The rest of the section is devoted to the proof of Theorem 4.4.

As in Section 4.2, we assume that transducer $\mathcal{T}$ outputs at most one letter on each transition, and that it writes $\#v$ with $v \in \Gamma^*$ when running on $\#u$ with $u \in \Sigma^*$. Using Lemma 4.5, we also assume that transitions $t' = (q', a', \varphi', \mathsf{op}', s')$ of $\mathcal{T}'$ do not both drop/lift a pebble and move the head: if $\mathsf{op}' \neq \mathsf{nop}$ then $s' \in Q'_0$.

We fix $u \in \mathsf{dom}(f) \subseteq \Sigma^*$ and $v = f(u) \in \Gamma^*$. We consider the accepting run of $\mathcal{T}$ on $\#u$ producing $\#v$:

$$C_0 \xrightarrow{t_0} C_1 \xrightarrow{t_1} C_2 \xrightarrow{t_2} \cdots C_N \xrightarrow{t_N} C_{N+1} \tag{3}$$

where $C_\ell = (q_\ell, \mathsf{peb}_\ell, \mathsf{h}_\ell)$ are configurations and $t_\ell \in \delta$ are transitions of $\mathcal{T}$.

As in Section 4.2, for each position $i \in \mathsf{epos}(v)$, we let $\mathsf{encode}(i)$ be the index of the transition in the above run producing the letter at position i in $\#v$. Again, the main idea is to encode a position $i \in \mathsf{epos}(v)$ by the configuration $C_{\mathsf{encode}(i)}$. Since $\mathcal{T}'$ has pebbles, this encoding is used not only for the head h' of $\mathcal{T}'$, but also for the pebbles dropped by $\mathcal{T}'$.

Recall that the ℓ-th configuration in (3) is $C_\ell = (q_\ell, \mathsf{peb}_\ell, \mathsf{h}_\ell)$. Consider a configuration $C' = (q', \mathsf{peb}', \mathsf{h}')$ of $\mathcal{T}'$ on $\#v$. Let $k = |\mathsf{peb}'|$, $j_i = \mathsf{encode}(\mathsf{peb}'_i)$ for $1 \leq i \leq k$, and $j = \mathsf{encode}(\mathsf{h}')$. Let $[n+1] = \{1, \ldots, n+1\}$. Define $\mathsf{encode}(C') = C'' = (q'', \mathsf{peb}'', \mathsf{h}'')$ where

$$q'' = (q_j, q', (q_{j_i})_{1 \leq i \leq k}, (1 + |\mathsf{peb}_{j_i}|)_{1 \leq i \leq k}) \in Q \times Q' \times Q^k \times [n+1]^k \subseteq Q''_0$$
$$\mathsf{peb}'' = \mathsf{peb}_{j_1} \mathsf{h}_{j_1} \cdots \mathsf{peb}_{j_k} \mathsf{h}_{j_k} \mathsf{peb}_j \qquad \text{and} \qquad \mathsf{h}'' = \mathsf{h}_j\,.$$

The set Q''_0 of 0-states of $\mathcal{T}''$ contains $\bigcup_{k=0}^{m} Q \times Q' \times Q^k \times [n+1]^k$. The initial and final states are $q''_i = (q_i, q'_i, (), ())$ and $q''_f = (q_i, q'_f, (), ())$ respectively. It remains to define the other states in Q'', the transition function δ'' and the output function μ''.

We explain below how a transition $C' \xrightarrow{t'} C'_1$ of $\mathcal{T}'$ will be simulated in $\mathcal{T}''$ by a sequence of transitions of the form $\mathsf{encode}(C') \xrightarrow{+} \mathsf{encode}(C'_1)$.

Consider a state $q'' = (q, q', \overline{x}, \overline{y}) \in Q \times Q' \times Q^k \times [n+1]^k$ for some $0 \leq k \leq m$, with $\overline{x} = (x_i)_{1 \leq i \leq k}$ and $\overline{y} = (y_i)_{1 \leq i \leq k}$. Let $t = (q, a, \varphi, \mathsf{op}, s)$ be a transition of $\mathcal{T}$ which produces an output letter $a' = \mu(t) \in \Gamma$. Consider a transition $t' = (q', a', \varphi', \mathsf{op}', s')$ of $\mathcal{T}'$ which reads the output produced by t. The goal is to synchronize the pair of transitions (t, t') as we did in Section 4.2. We first explain how to write a test ξ checking whether both t and t' are "enabled" at a configuration $C'' = (q'', \mathsf{peb}'', \mathsf{h})$ of $\mathcal{T}''$. This is intended to be used in particular when C'' is of the form $\mathsf{encode}(C')$ for some configuration C' of $\mathcal{T}'$.

We introduce some notation. For each $0 \leq \ell \leq k$, we let $d_\ell = y_1 + \cdots + y_\ell$ which can be recovered from the state q'' of $\mathcal{T}''$. Given an offset $d \geq 0$ and a test φ of $\mathcal{T}$, we write φ^{+d} the test obtained by adding d to the pebble indices: $\mathsf{p}_i = \mathsf{p}_j$ is replaced with $\mathsf{p}_{i+d} = \mathsf{p}_{j+d}$ and $\mathsf{h} = \mathsf{p}_j$ is replaced with $\mathsf{h} = \mathsf{p}_{j+d}$. Finally, let $\xi_0 = (\mathsf{p}_0 = \mathsf{p}_0) \wedge \neg(\mathsf{p}_{n+1} = \mathsf{p}_{n+1})$ which will be used, shifted by some offset, to check the size of the pebble stack.

▷ Claim 4.6. Let $C'' = (q'', \mathsf{peb}'', \mathsf{h})$ be a configuration of $\mathcal{T}''$ with $q'' = (q, q', \overline{x}, \overline{y})$. Let ψ be a test of $\mathcal{T}$ and ψ' be a test of $\mathcal{T}'$.

1. $\mathsf{peb}'', \mathsf{h} \models \xi_0^{+d_k}$ if and only if $d_k \leq |\mathsf{peb}''| \leq d_k + n$.
   We assume below that we are in this case and we write $\mathsf{peb}'' = \mathsf{peb}^1\mathsf{h}^1 \cdots \mathsf{peb}^k\mathsf{h}^k\mathsf{peb}$ with $|\mathsf{peb}^\ell\mathsf{h}^\ell| = y_\ell$ for $1 \leq \ell \leq k$ and $0 \leq |\mathsf{peb}| \leq n$.
2. $\mathsf{peb}'', \mathsf{h} \models \psi^{+d_k}$ if and only if $\mathsf{peb}, \mathsf{h} \models \psi$.
3. We can construct a formula $\overline{\xi}(q'', \psi')$ such that, for all $\mathsf{peb}', \mathsf{h}'$ with $|\mathsf{peb}'| = k$ and (a) $\mathsf{peb}'_i = \mathsf{peb}'_j$ iff $x_i = x_j$, $\mathsf{peb}^i = \mathsf{peb}^j$ and $\mathsf{h}^i = \mathsf{h}^j$; and (b) $\mathsf{h}' = \mathsf{peb}'_j$ iff $q = x_j$, $\mathsf{peb} = \mathsf{peb}^j$ and $\mathsf{h} = \mathsf{h}^j$, we have $\mathsf{peb}'', \mathsf{h} \models \overline{\xi}(q'', \psi')$ if and only if $\mathsf{peb}', \mathsf{h}' \models \psi'$.

Proof. Items 1 and 2 are clear. For Item 3 we let $\overline{\xi}(q'', \psi')$ be the formula ψ' in which we replace each atom of the form $\mathsf{h}' = \mathsf{p}'_i$ by

- false if $i > k$ or $x_i \neq q$,
- $\left(\bigwedge_{\ell=1}^{y_i - 1} \mathsf{p}_{\ell+d_k} = \mathsf{p}_{\ell+d_i-1}\right) \wedge (\mathsf{h} = \mathsf{p}_{d_i}) \wedge \neg(\mathsf{p}_{y_i+d_k} = \mathsf{p}_{y_i+d_k})$ otherwise;

and each atom of the form $\mathsf{p}'_i = \mathsf{p}'_j$ by

- false if $i > k$ or $j > k$ or $x_i \neq x_j$ or $y_i \neq y_j$,
- $\bigwedge_{\ell=1}^{y_i} \mathsf{p}_{\ell+d_{i-1}} = \mathsf{p}_{\ell+d_j-1}$ otherwise. ◁

Let $\xi = (\xi_0 \wedge \varphi \wedge \overline{\mathsf{op}})^{+d_k} \wedge \overline{\xi}(q'', \varphi' \wedge \overline{\mathsf{op}'})$. From Claim 4.6, assuming that $C'' = \mathsf{encode}(C')$, we see that $C'' \models \xi$ if and only if t is enabled at $C = (q, \mathsf{peb}, \mathsf{h})$ and t' is enabled at $C' = (q', \mathsf{peb}', \mathsf{h}')$. We explain now how to synchronize the pair of transitions (t, t') from state q''. Given an offset $d \geq 0$ and an operation op of $\mathcal{T}$, we write op^{+d} the operation shifted by d: $\mathsf{drop}_i^{+d} = \mathsf{drop}_{i+d}$, $\mathsf{lift}_i^{+d} = \mathsf{lift}_{i+d}$ and $\mathsf{nop}^{+d} = \mathsf{nop}$.

Case nop. We have $t' = (q', a', \varphi', \mathsf{nop}, s')$ with $s' \in Q'$. We check whether the pair (t, t') is enabled with ξ and we implement the move induced by s' of the head of $\mathcal{T}'$ on $\#v$ as we did in Section 4.2. To do so, we introduce the following transition with $\mu''(t'') = \mu'(t')$:

$$t'' = (q, q', \overline{x}, \overline{y}) \xrightarrow{a, \xi, \mathsf{op}^{+d_k}} (\hat{s}, s', \overline{x}, \overline{y}) \qquad \text{if } s' \in Q'_{+1} \qquad \text{(gtr-a)}$$

$$t'' = (q, q', \overline{x}, \overline{y}) \xrightarrow{a, \xi, \mathsf{nop}} (\hat{q}, s', \overline{x}, \overline{y}) \qquad \text{if } s' \in Q'_{-1} \qquad \text{(gtr-b)}$$

$$t'' = (q, q', \overline{x}, \overline{y}) \xrightarrow{a, \xi, \mathsf{nop}} (q, s', \overline{x}, \overline{y}) \qquad \text{if } s' \in Q'_0 \qquad \text{(gtr-c)}$$

Here, $\widehat{Q} = \{\hat{q} \mid q \in Q\}$ is a disjoint copy of Q. The polarity of a simulation state $(\hat{q}, q', \overline{x}, \overline{y}) \in \widehat{Q} \times (Q'_{-1} \cup Q'_{+1}) \times Q^k \times [n+1]^k$ with $0 \le k \le m$ is the product of the polarity of q and the polarity of q'. When in a simulation state $(\hat{q}, q', \overline{x}, \overline{y})$, we simulate $\mathcal{T}$ forward or backward depending on the polarity of q' using transitions producing ε until we reach a transition producing a symbol from Γ. For each transition $t = (q, a, \varphi, \mathsf{op}, s)$ of $\mathcal{T}$ with $\mu(t) = \varepsilon$ we introduce the following transition t'' in $\mathcal{T}''$ with $\mu''(t'') = \varepsilon$:

$$t'' = (\hat{q}, q', \overline{x}, \overline{y}) \xrightarrow{a, \varphi^{+d_k}, \mathsf{op}^{+d_k}} (\hat{s}, q', \overline{x}, \overline{y}) \qquad \text{if } q' \in Q'_{+1} \qquad \text{(gmv-a)}$$

$$t'' = (\hat{s}, q', \overline{x}, \overline{y}) \xrightarrow{a, ((\varphi, \mathsf{op})^r)^{+d_k}} (\hat{q}, q', \overline{x}, \overline{y}) \qquad \text{if } q' \in Q'_{-1} \qquad \text{(gmv-b)}$$

and for each transition $t = (q, a, \varphi, \mathsf{op}, s)$ of $\mathcal{T}$ with $\mu(t) \in \Gamma$ we introduce the following transition t'' in $\mathcal{T}''$ with $\mu''(t'') = \varepsilon$ in order to switch back from the simulation mode to the synchronization mode:

$$t'' = (\hat{q}, q', \overline{x}, \overline{y}) \xrightarrow{a, (\varphi \wedge \overline{\mathsf{op}})^{+d_k}, \mathsf{nop}} (q, q', \overline{x}, \overline{y}) \qquad \text{if } q' \in Q'_{+1} \qquad \text{(gsw-a)}$$

$$t'' = (\hat{s}, q', \overline{x}, \overline{y}) \xrightarrow{a, ((\varphi, \mathsf{op})^r)^{+d_k}} (q, q', \overline{x}, \overline{y}) \qquad \text{if } q' \in Q'_{-1} \qquad \text{(gsw-b)}$$

As in the proof of Theorem 4.1, we can show that $\mathcal{T}''$ is reversible at simulation states of the form $(\hat{q}, q', \overline{x}, \overline{y})$. We can also prove similarly the following analog of Claim 4.2.

▷ Claim 4.7. Assuming that the operation of t' is nop, there is a transition of $\mathcal{T}'$ $C' = (q', \mathsf{peb}', \mathsf{h}') \xrightarrow{t'} C'_1 = (s', \mathsf{peb}', \mathsf{h}' + s')$ on $\#v$ if and only if there is a nonempty run $\mathsf{encode}(C') \xrightarrow{+} \mathsf{encode}(C'_1)$ of $\mathcal{T}''$ on $\#u$ where intermediate states are simulation states.

Case lift. We have $t' = (q', a', \varphi', \mathsf{lift}_{k'}, s')$ with $1 \le k' \le m$ and $s' \in Q'_0$ from our assumption on $\mathcal{T}'$ (Lemma 4.5). We still check that the pair (t, t') is enabled with $\xi = (\xi_0 \wedge \varphi \wedge \overline{\mathsf{op}})^{+d_k} \wedge \overline{\xi}(q'', \varphi' \wedge \overline{\mathsf{lift}_{k'}})$. Thanks to $\overline{\xi}(q'', \overline{\mathsf{lift}_{k'}})$, we get $k' = k$, $q = x_k$, $\mathsf{peb}^k = \mathsf{peb}$ and $\mathsf{h}^k = \mathsf{h}$. We use the lift-gadget given in Figure 3 to pop the top y_k pebbles, i.e., $\mathsf{h}^k\mathsf{peb}$ (in reverse order). We enter and leave the lift-gadget with the following transitions where $s'' = (q, s', (x_1, \ldots, x_{k-1}), (y_1, \ldots, y_{k-1})) \in Q''_0$, $\xi' = (\xi_0 \wedge \varphi \wedge \overline{\mathsf{op}})^{+(d_k - y_k)} \wedge \overline{\xi}(s'', \mathsf{op}'(\varphi') \wedge \overline{\mathsf{op}'^r})$, $\mu''(t''_{\mathsf{first}}) = \mu'(t')$ and $\mu''(t''_{\mathsf{last}}) = \varepsilon$:

$$t''_{\mathsf{first}} = q'' \xrightarrow{a, \xi, \mathsf{nop}} (q'', -1) \qquad \text{(lift-a)}$$

$$t''_{\mathsf{last}} = (q'', 0) \xrightarrow{a, \xi', \mathsf{nop}} s'' \qquad \text{(lift-b)}$$

The blue tests are added to ensure reverse determinism at the internal states. Also, when the before last transition is taken we see p_{d_k}. Hence, the head is on the same position at the beginning and at the end of the gadget.

Case drop. The operation of transition t' is $\mathsf{op}' = \mathsf{drop}_{k'}$ $(1 \le k' \le m)$. We still check that the pair (t, t') is enabled with $\xi = (\xi_0 \wedge \varphi \wedge \overline{\mathsf{op}})^{+d_k} \wedge \overline{\xi}(q'', \varphi' \wedge \overline{\mathsf{drop}_{k'}})$. Thanks to $\overline{\xi}(q'', \overline{\mathsf{drop}_{k'}})$, we get $k' = k + 1$. We use the drop-gadget given in Figure 4 to push h and then peb (in this order) on the pebble stack. We enter and leave the drop-gadget with the following transitions where $1 \le z \le n+1$, $s'' = (q, s', (x_1, \ldots, x_k, q), (y_1, \ldots, y_k, z)) \in Q''_0$, $\xi' = (\xi_0 \wedge \varphi \wedge \overline{\mathsf{op}})^{+(d_k + z)} \wedge \overline{\xi}(s'', \mathsf{op}'(\varphi') \wedge \overline{\mathsf{op}'^r})$, $\mu''(t''_{\mathsf{first}}) = \mu'(t')$ and $\mu''(t''_{\mathsf{last}}) = \varepsilon$:

$$t''_{\mathsf{first}} = q'' \xrightarrow{a, \xi, \mathsf{drop}_{d_k + z}} (q'', z, 1) \qquad \text{(drop-a)}$$

$$t''_{\mathsf{last}} = (q'', z, z) \xrightarrow{a, \xi', \mathsf{nop}} s'' \qquad \text{(drop-b)}$$

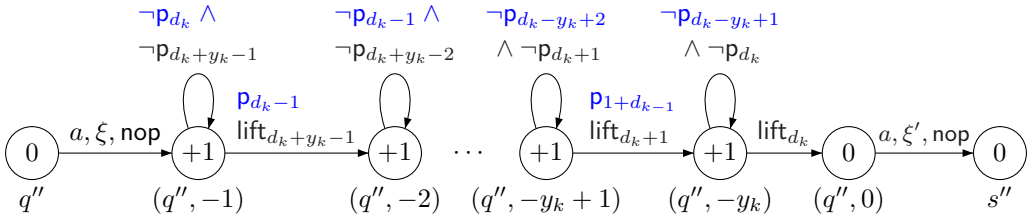


Figure 3 Simulation of lift_k. The first state is $q'' = (q, q', (x_1, \dots, x_k), (y_1, \dots, y_k)) \in Q''_0$ and $d_k = y_1 + \cdots + y_k$. The internal states are of the from $(q'', -\ell) \in Q''_{+1}$ for $0 \le \ell \le y_k$. The last state is $s'' = (q, s', (x_1, \dots, x_{k-1}), (y_1, \dots, y_{k-1})) \in Q''_0$. The test on the first transition is $\xi = (\xi_0 \wedge \varphi \wedge \overline{\mathsf{op}})^{+d_k} \wedge \overline{\xi}(q'', \varphi' \wedge \overline{\mathsf{op}'})$. The test on the last transition is $\xi' = (\xi_0 \wedge \varphi \wedge \overline{\mathsf{op}})^{+(d_k - y_k)} \wedge \overline{\xi}(s'', \mathsf{op}'(\varphi') \wedge \overline{\mathsf{op}'^r})$.

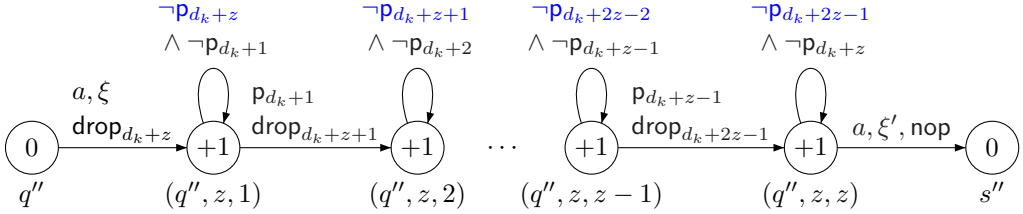


Figure 4 Simulation of drop_{k+1} with z any number such that $1 \le z \le n+1$.
The first state is $q'' = (q, q', (x_1, \dots, x_k), (y_1, \dots, y_k)) \in Q''_0$ and $d_k = y_1 + \cdots + y_k$. The internal states are of the from $(q'', z, \ell) \in Q''_{+1}$ for $1 \le \ell \le z$. The last state is $s'' = (q, s', (x_1, \dots, x_k, q), (y_1, \dots, y_k, z)) \in Q''_0$. The test on the first transition is $\xi = (\xi_0 \wedge \varphi \wedge \overline{\mathsf{op}})^{+d_k} \wedge \overline{\xi}(q'', \varphi' \wedge \overline{\mathsf{op}'})$. The test on the last transition is $\xi' = (\xi_0 \wedge \varphi \wedge \overline{\mathsf{op}})^{+(d_k+z)} \wedge \overline{\xi}(s'', \mathsf{op}'(\varphi') \wedge \overline{\mathsf{op}'^r})$.

On the first transition of the gadget, the test ξ makes sure that the pebble stack is of the form $\mathsf{peb}'' = \mathsf{peb}^1\mathsf{h}^1 \cdots \mathsf{peb}^k\mathsf{h}^k\mathsf{peb}$ with $|\mathsf{peb}^\ell\mathsf{h}^\ell| = y_\ell$ for $1 \le \ell \le k$ and $0 \le |\mathsf{peb}| \le n$. The operation drop_{d_k+z} on the first transition is only possible if $z = |\mathsf{peb}| + 1$ and it allows to determine the size of peb. Since $\mathsf{op}'^r = \mathsf{lift}_{k+1}$, the test ξ' on the last transition contains $\overline{\xi}(s'', \overline{\mathsf{lift}_{k+1}})$ which contains the test p_{d_k+z}. This test makes sure that at the end of the gadget, the head of $\mathcal{T}''$ is on the same position of the input word $\#u$ as it was at the beginning. The drop gadget is deterministic at the internal states. The blue tests are added to make this gadget reverse deterministic at the internal states.

We claim that $\mathcal{T}''$ is deterministic, and that if $\mathcal{T}'$ is reverse-deterministic then $\mathcal{T}''$ is reverse-deterministic (see [11] for detailed proofs).

5 Generators for Pebble Transducers

Set of generators. In [5], the class of polyregular functions is defined as the smallest class of functions closed under composition that contains the sequential functions, the squaring function and the iterated reverse function. The iterated reverse function acts on an alphabet Σ enriched with a special symbol ! and maps a word $u_0!u_1!\dots!u_n$ to $u_0^r!u_1^r!\dots!u_n^r$. By proving that each of these generators can easily be realized by a reversible pebble transducer and using Theorem 4.4, we prove that reversible pebble transducers realize all polyregular functions. This also gives a way to generate them using these basic blocks and composition.

Using [10, Theorem 2] as well as Remark 2.1, we get that any sequential function, realized by a transducer T with n states, can be realized by a reversible 0-pebble transducer with $\mathcal{O}(n^2)$ states. A reversible 1-pebble transducer for the squaring function was given in Example 2.2. Finally, we give a reversible 0-pebble transducer for the iterated reverse in Figure 5.

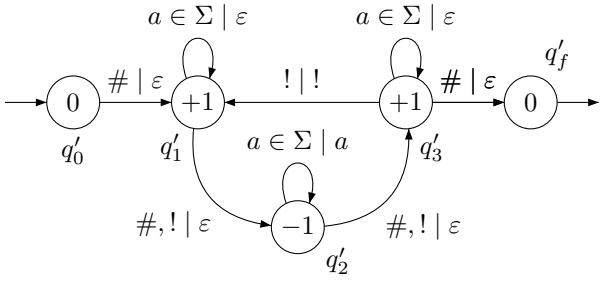


Figure 5 A 0-pebble transducer realizing the iterated reverse function. As there is no pebble, we omitted the tests and the pebble actions.

Uniformizing Pebble transducers. Another way to generate reversible pebble transducers is to start from a possibly non deterministic pebble transducer and uniformize it by a reversible pebble transducer. This section provides a procedure to do this, while preserving the number of pebbles used by the given transducer. By uniformizing a relation R, we mean extract a function f such that $\mathsf{dom}(f) = \mathsf{dom}(R)$ and $f \subseteq R$.

▶ **Theorem 5.1.** *Given a k-pebble transducer $\mathcal{T}$ with n states (possibly with equality tests), one can construct a k-pebble reversible transducer R with $2^{\mathcal{O}((kn)^2)}$ states such that $[\![R]\!]$ is a uniformization of $[\![\mathcal{T}]\!]$.*

Proof. The proof relies on the composition of reversible pebble transducers, the uniformization result from [10] and lemmas formally stated and proven in Appendix B. $\mathcal{T}$ can be decomposed into a basic reversible k-pebble transducer $\mathcal{C}_k$, a reversible 0-pebble transducer $\mathcal{C}_k^=$ enabling equality tests, and a 0-pebble transducer $\mathcal{T}_0$ with respectively $\mathcal{O}(k)$, $\mathcal{O}(2^{k^2})$ and $\mathcal{O}(kn)$ states. The transducer $\mathcal{C}_k$ associates to a word u the word of length $|\#u|^{|\#u|^k}$ that is the sequence of possible configurations of a k pebble transducer over the word u. The transducer $\mathcal{C}_k^=$ adds the truth values of equality tests to the configurations. The 0-pebble transducer $\mathcal{T}_0$ uses this information to simulate $\mathcal{T}$. Then, by Remark 2.1, $\mathcal{T}_0$ is transformed into a two-way transducer $\mathcal{T}'_0$ with $\mathcal{O}(kn)$ states. We can then use [10, Theorem 4] to obtain a reversible two-way transducer RT with $2^{\mathcal{O}((kn)^2)}$ states. Using Remark 2.1 back, RT is transformed into a 0-pebble transducer RP with $2^{\mathcal{O}((kn)^2)}$ states. We can conclude by composing $\mathcal{C}_k$ with $\mathcal{C}_k^=$ and composing the result with RP using Theorem 4.1 twice. We obtain a reversible k-pebble transducer R with $2^{\mathcal{O}((kn)^2)}$ states. ◀

References

1 Rajeev Alur and Pavol Černý. Expressiveness of streaming string transducers. In *30th International Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2010*, volume 8 of *LIPIcs. Leibniz Int. Proc. Inform.*, pages 1–12. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2010. doi:10.4230/LIPICS.FSTTCS.2010.1.

2 Rajeev Alur, Adam Freilich, and Mukund Raghothaman. Regular combinators for string transformations. In Thomas A. Henzinger and Dale Miller, editors, *Joint Meeting of the 23rd EACSL Annual Conference on Computer Science Logic (CSL) and the 29th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014*, pages 9:1–9:10. ACM, 2014. doi:10.1145/2603088.2603151.

3 Nicolas Baudru and Pierre-Alain Reynier. From two-way transducers to regular function expressions. In Mizuho Hoshi and Shinnosuke Seki, editors, *22nd International Conference on Developments in Language Theory, DLT 2018*, volume 11088 of *Lecture Notes in Computer Science*, pages 96–108. Springer, 2018. doi:10.1007/978-3-319-98654-8_8.

4 Mikolaj Bojanczyk. Polyregular functions. *CoRR*, abs/1810.08760, 2018. doi:10.48550/arXiv.1810.08760.

5 Mikolaj Bojanczyk. Transducers of polynomial growth. In Christel Baier and Dana Fisman, editors, *LICS '22: 37th Annual ACM/IEEE Symposium on Logic in Computer Science, Haifa, Israel, August 2 - 5, 2022*, pages 1:1–1:27. ACM, 2022. doi:10.1145/3531130.3533326.

6 Mikolaj Bojanczyk, Laure Daviaud, and Shankara Narayanan Krishna. Regular and first-order list functions. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 125–134, 2018. doi:10.1145/3209108.3209163.

7 Mikolaj Bojanczyk, Sandra Kiefer, and Nathan Lhote. String-to-string interpretations with polynomial-size output. In Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini, and Stefano Leonardi, editors, *46th International Colloquium on Automata, Languages, and Programming, ICALP 2019, July 9-12, 2019, Patras, Greece*, volume 132 of *LIPIcs*, pages 106:1–106:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPICS.ICALP.2019.106.

8 Michal P. Chytil and Vojtěch Jákl. Serial composition of 2-way finite-state transducers and simple programs on strings. In *Automata, languages and programming (Fourth Colloq., Univ. Turku, Turku, 1977)*, pages 135–137. Lecture Notes in Comput. Sci., Vol. 52. Springer, Berlin, 1977.

9 Bruno Courcelle. Monadic second-order definable graph transductions: a survey [see MR1251992 (94f:68009)]. *Theoret. Comput. Sci.*, 126(1):53–75, 1994. Seventeenth Colloquium on Trees in Algebra and Programming (CAAP '92) and European Symposium on Programming (ESOP) (Rennes, 1992). doi:10.1016/0304-3975(94)90268-2.

10 Luc Dartois, Paulin Fournier, Ismaël Jecker, and Nathan Lhote. On reversible transducers. In Ioannis Chatzigiannakis, Piotr Indyk, Fabian Kuhn, and Anca Muscholl, editors, *44th International Colloquium on Automata, Languages, and Programming, ICALP 2017, Warsaw, Poland*, volume 80 of *LIPIcs*, pages 113:1–113:12. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.ICALP.2017.113.

11 Luc Dartois, Paul Gastin, Loïc Germerie Guizouarn, and Shankaranarayanan Krishna. Reversible pebble transducers. *CoRR*, 2025. doi:10.48550/arXiv.2506.11334.

12 Vrunda Dave, Paul Gastin, and Shankara Narayanan Krishna. Regular Transducer Expressions for Regular Transformations. In Martin Hofmann, Anuj Dawar, and Erich Grädel, editors, *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic In Computer Science (LICS'18)*, pages 315–324, Oxford, UK, July 2018. ACM Press. doi:10.1145/3209108.3209182.

13 Joost Engelfriet. Two-way pebble transducers for partial functions and their composition. *Acta Informatica*, 52(7-8):559–571, 2015. doi:10.1007/S00236-015-0224-3.

14 Joost Engelfriet and Hendrik Jan Hoogeboom. MSO definable string transductions and two-way finite-state transducers. *ACM Transactions on Computational Logic*, 2(2):216–254, 2001. doi:10.1145/371316.371512.

15 Joost Engelfriet and Sebastian Maneth. Two-way finite state transducers with nested pebbles. In Krzysztof Diks and Wojciech Rytter, editors, *Mathematical Foundations of Computer Science 2002, 27th International Symposium, MFCS 2002, Warsaw, Poland, August 26-30, 2002, Proceedings*, volume 2420 of *Lecture Notes in Computer Science*, pages 234–244. Springer, 2002. doi:10.1007/3-540-45687-2_19.

A Composition of pebble transducers

A.1 Composition: Simple case

▷ Claim 4.2. There is a transition $(q', \mathsf{h}') \xrightarrow{t'} (s', \mathsf{h}' + s')$ of $\mathcal{T}'$ on $\#v$ if and only if there is a nonempty run $\mathsf{encode}((q', \mathsf{h}')) \xrightarrow{+} \mathsf{encode}((s', \mathsf{h}' + s'))$ of $\mathcal{T}''$ on $\#u$ which does not use an intermediate state in $Q \times Q'$.

Proof. ($\Longrightarrow$) Assume that $\mathcal{T}''$ has a transition $(q', \mathsf{h}') \xrightarrow{t'} (s', \mathsf{h}' + s')$ on $\#v$. Let a' be the letter of $\#v$ at position h'. Let $\ell = \mathsf{encode}(\mathsf{h}')$ and $\ell' = \mathsf{encode}(\mathsf{h}' + s')$. We have $\mu(t_\ell) = a'$ and $t' = (q', a', s')$. Let t''_ℓ be the transition of $\mathcal{T}''$ constructed from the pair (t, t') using Equations (tr-a)–(tr-c).

If $s' \in Q'_0$ then $\ell' = \ell$ and $\mathsf{encode}((q', \mathsf{h}')) \xrightarrow{t''_\ell} \mathsf{encode}((s', \mathsf{h}' + s'))$: we are done.

Assume that $s' \in Q'_{+1}$. Then $\ell' > \ell$.[2] For each $\ell < i < \ell'$, let t''_i be the transition of $\mathcal{T}''$ constructed from t_i by (mv-a). Let $t''_{\ell'}$ be the transition of $\mathcal{T}''$ constructed from $t_{\ell'}$ by (sw-a). We can easily check that

$$((q_\ell, q'), \mathsf{peb}_\ell, \mathsf{h}_\ell) \xrightarrow{t''_\ell} ((q_{\ell\hat{+}1}, s'), \mathsf{peb}_{\ell+1}, \mathsf{h}_{\ell+1}) \xrightarrow{t''_{\ell+1} \cdots t''_{\ell'-1}} ((q_{\hat{\ell'}}, s'), \mathsf{peb}_{\ell'}, \mathsf{h}_{\ell'})$$
$$\cdots \xrightarrow{t''_{\ell'}} ((q_{\ell'}, s'), \mathsf{peb}_{\ell'}, \mathsf{h}_{\ell'})$$

Therefore, $\mathsf{encode}((q', \mathsf{h}')) \xrightarrow{t''_\ell t''_{\ell+1} \cdots t''_{\ell'-1} t''_{\ell'}} \mathsf{encode}((s', \mathsf{h}' + s'))$.

The third case is $s' \in Q'_{-1}$. Then $\ell' < \ell$.[3] For each $\ell' < i < \ell$, let t''_i be the transition of $\mathcal{T}''$ constructed from t_i by (mv-b). Let $t''_{\ell'}$ be the transition of $\mathcal{T}''$ constructed from $t_{\ell'}$ by (sw-b). We can easily check that

$$((q_\ell, q'), \mathsf{peb}_\ell, \mathsf{h}_\ell) \xrightarrow{t''_\ell} ((\hat{q}_\ell, s'), \mathsf{peb}_\ell, \mathsf{h}_\ell - q_\ell) \xrightarrow{t''_{\ell-1}} ((q_{\ell\hat{-}1}, s'), \mathsf{peb}_{\ell-1}, \mathsf{h}_{\ell-1} - q_{\ell-1})$$
$$\cdots \xrightarrow{t''_{\ell'+1}} ((q_{\ell'\hat{+}1}, s'), \mathsf{peb}_{\ell'+1}, \mathsf{h}_{\ell'+1} - q_{\ell'+1}) \xrightarrow{t''_{\ell'}} ((q_{\ell'}, s'), \mathsf{peb}_{\ell'}, \mathsf{h}_{\ell'} - 0)$$

We deduce that $\mathsf{encode}((q', \mathsf{h}')) \xrightarrow{t''_\ell t''_{\ell-1} \cdots t''_{\ell'+1} t''_{\ell'}} \mathsf{encode}((s', \mathsf{h}' + s'))$ as desired.

($\Longleftarrow$) The converse can be shown similarly by noting that, depending on the polarity of s', in a run $\mathsf{encode}((q', \mathsf{h}')) \xrightarrow{+} \mathsf{encode}((s', \mathsf{h}' + s'))$ which has no intermediate states in $Q \times Q'$, the sequence of transitions used is one of

(a) a transition constructed with (tr-a), followed by a (possibly empty) sequence of transitions constructed with (mv-a), followed by a transition constructed with (sw-a),

(b) a transition constructed with (tr-b), followed by a (possibly empty) sequence of transitions constructed with (mv-b), followed by a transition constructed with (sw-b),

(c) a single transition constructed with (tr-c). ◁

▷ Claim A.1. The transducer $\mathcal{T}''$ from Theorem 4.1 is deterministic.

Proof. Consider two transitions t''_1 and t''_2 starting from some state q'', reading some $a \in \Sigma$ and both enabled at some configuration $C'' = (q'', \mathsf{peb}, \mathsf{h})$ of $\mathcal{T}''$.

Assume first that $q'' = (\hat{q}, q')$ with $q' \in Q'_{+1}$. For each $i \in \{1, 2\}$, let $t_i = (q, a, \varphi_i, \mathsf{op}_i, s_i)$ be the transition of $\mathcal{T}$ giving rise to t''_i with (mv-a) or (sw-a). Since t''_i is enabled at configuration C'', we deduce that t_i is enabled at configuration $C = (q, \mathsf{peb}, \mathsf{h})$. Hence, t_1, t_2 are both enabled at C. We get $t_1 = t_2$ by determinism of $\mathcal{T}$ and therefore $t''_1 = t''_2$.

Assume now that $q'' = (\hat{s}, q')$ with $q' \in Q'_{-1}$. For each $i \in \{1, 2\}$, let $t_i = (q_i, a, \varphi_i, \mathsf{op}_i, s)$ be the transition of $\mathcal{T}$ giving rise to t''_i with (mv-b) or (sw-b). Since t''_i is enabled at configuration C'', we deduce that t^r_i is enabled at configuration $C = (s, \mathsf{peb}, \mathsf{h})$. Therefore, by (1), transition t_i is reverse-enabled at configuration $C' = (s, \mathsf{peb}, \mathsf{h} + s)$. Hence, t_1, t_2 are both reverse-enabled at C'. We get $t_1 = t_2$ by reverse-determinism of $\mathcal{T}$. Hence, $t''_1 = t''_2$.

---

[2] Or $\ell = |v|$ and $\ell' = 0$ which can be handled similarly using the extra transition $t_{f,i}$ producing ε allowing to move from the final configuration of $\mathcal{T}$ to its initial configuration.

[3] Or $\ell = 0$ and $\ell' = |v|$ which can be handled similarly using the extra transition $t_{f,i}$.

Finally, assume that $q'' = (q, q')$. For each $i \in \{1, 2\}$, let $t_i = (q, a, \varphi_i, \mathsf{op}_i, s_i)$, $a'_i = \mu(t_i)$ and $t'_i = (q', a'_i, s'_i)$ be the transitions of $\mathcal{T}$ and $\mathcal{T}'$ giving rise to t''_i with (tr-a), (tr-b) or (tr-c). Since t''_i is enabled at C'', we deduce that t_i is enabled at configuration $C = (q, \mathsf{peb}, \mathsf{h})$. Hence, t_1, t_2 are both enabled at C and we get $t_1 = t_2$ by determinism of $\mathcal{T}$. It follows that $a'_1 = \mu(t_1) = \mu(t_2) = a'_2$ and we get $t'_1 = t'_2$ by determinism of $\mathcal{T}'$. Therefore $t''_1 = t''_2$. ◁

▷ Claim A.2. If the transducer $\mathcal{T}'$ from Theorem 4.1 is reversible, then $\mathcal{T}''$ is reverse-deterministic.

Proof. Consider two transitions t''_1 and t''_2 of $\mathcal{T}''$ ending in some state q'', reading some $a \in \Sigma$ and both reverse-enabled at some configuration $C'' = (q'', \mathsf{peb}, \mathsf{h})$.

Assume first that $q'' = (\hat{s}, s')$ with $s' \in Q'_{+1}$. For each $i \in \{1, 2\}$, let $t_i = (q_i, a, \varphi_i, \mathsf{op}_i, s)$ be the transition of $\mathcal{T}$ giving rise to t''_i with (mv-a) or (tr-a). Since t''_i is reverse-enabled at configuration C'', we deduce that t_i is reverse-enabled at configuration $C = (s, \mathsf{peb}, \mathsf{h})$. Hence, t_1, t_2 are both reverse-enabled at C and we get $t_1 = t_2 = t$ by reverse-determinism of $\mathcal{T}$. Now, either $\mu(t) = \varepsilon$ and both t''_1 and t''_2 are constructed from t and s' with (mv-a). We get $t''_1 = t''_2$. Or $\mu(t) = a' \in \Gamma$ and t''_i is constructed from t and some transition $t'_i = (q'_i, a', s')$ of $\mathcal{T}'$ with (tr-a). Using the reverse-determinism of $\mathcal{T}'$, we deduce that $t'_1 = t'_2$ and $t''_1 = t''_2$.

Assume now that $q'' = (q, q') \in Q''_0$. Depending on the polarity of q', *both* transitions t''_1 and t''_2 are constructed with the *same* case (sw-a), (sw-b) or (tr-c) from some transitions t_1 and t_2 of $\mathcal{T}$. In the cases (sw-a) or (tr-c), the pebble stack does not change when executing t''_i whose operation is nop, and the head does not move since $q'' \in Q''_0$. Since t''_i is reverse-enabled at C'', we deduce that $\mathsf{peb}, \mathsf{h} \models \varphi_i \wedge \overline{\mathsf{op}_i}$. Hence, t_1 and t_2 are both enabled at configuration $C = (q, \mathsf{peb}, \mathsf{h})$. It follows that $t_1 = t_2$ using the determinism of $\mathcal{T}$, and then $t''_1 = t''_2$, using the reverse-determinism of $\mathcal{T}'$ in case (tr-c). Consider now the case (sw-b) and let $t_i = (q, a, \varphi_i, \mathsf{op}_i, s_i)$. Since t''_i is reversed-enabled at C'', we deduce that t^r_i is reverse-enabled at $C = (q, \mathsf{peb}, \mathsf{h})$. By (1), this implies that t_i is enabled at $(q, \mathsf{peb}, \mathsf{h} + q)$. We conclude as in the previous case that $t_1 = t_2$ and $t''_1 = t''_2$.

Finally, assume that $q'' = (\hat{q}, q') \in \widehat{Q} \times Q'_{-1}$. For each $i \in \{1, 2\}$, let $t_i = (q, a, \varphi_i, \mathsf{op}_i, s_i)$ be the transition of $\mathcal{T}$ giving rise to t''_i with (mv-b) or (tr-b). In case (mv-b), we see that t^r_i is reverse-enabled at $(q, \mathsf{peb}, \mathsf{h})$ and using (1) we deduce that t_i is enabled at $C = (q, \mathsf{peb}, \mathsf{h} + q)$. In case (tr-b), we note that the pebble stack does not change while executing transition t''_i and the head moves by the polarity of q''. Since t''_i is reverse-enabled at C'', we have $\mathsf{peb}, \mathsf{h} - q'' \models \varphi_i \wedge \overline{\mathsf{op}_i}$. Using $\mathsf{h} - q'' = \mathsf{h} + q$ we deduce that t_i is enabled at $C = (q, \mathsf{peb}, \mathsf{h} + q)$. We have shown that in both cases (mv-b) or (tr-b), transitions t_1 and t_2 are both enabled at C. It follows that $t_1 = t_2$ using the determinism of $\mathcal{T}$, and then $t''_1 = t''_2$, using the reverse-determinism of $\mathcal{T}'$ in case (tr-b). ◁

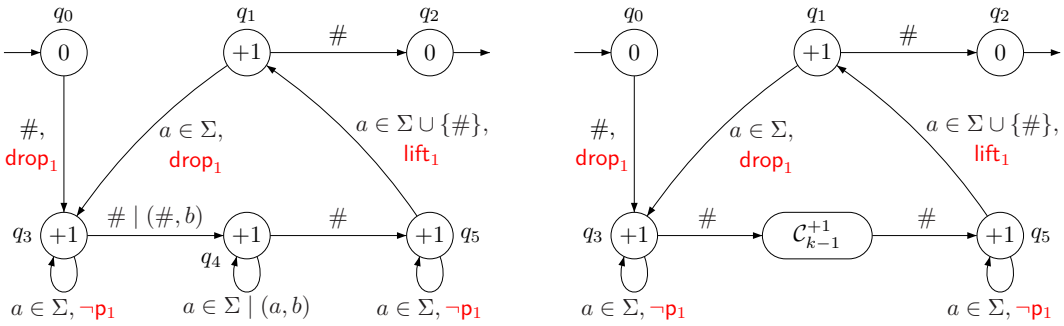


Figure 6 Reversible transducers for the function $\mathcal{C}_1$ (left) and $\mathcal{C}_k$ (right). The production b of $\mathcal{C}_1$ is the bit corresponding to $\mathsf{h} = \mathsf{p}_1$. The transitions are actually duplicated. The self-loop on q_4 reading $a \in \Sigma$ stands for two transitions $(q_4, a, \mathsf{p}_1, q_4)$ producing $(a, 1)$ and $(q_4, a, \neg\mathsf{p}_1, q_4)$ producing $(a, 0)$. Similarly, the transition from q_3 to q_4 reading # is duplicated.
In $\mathcal{C}_k$, we use a copy $\mathcal{C}_{k-1}^{+1}$ of $\mathcal{C}_{k-1}$ where all the pebble indices are incremented by 1. The transition in $\mathcal{C}_k$ from q_3 labelled # goes to the initial state of $\mathcal{C}_{k-1}^{+1}$ which is a 0-state, hence the head does not move. Similarly, the transition in $\mathcal{C}_k$ labelled # going to q_5 starts from the final state of $\mathcal{C}_{k-1}^{+1}$. When it appears in $\mathcal{C}_k$, b stands for the bit vector giving the truth value of each $\mathsf{h} = \mathsf{p}_i$ for $1 \leq i \leq k$. It is actually a macro for 2^k disjoint transitions.

B Generators for pebble transducers

We give the supporting lemmas and their proof required for Theorem 5.1.

Given an integer $k \geq 1$ and a word u, we define the marking of u for k-configurations as the word $\mathcal{C}_k(u)$ on $(\Sigma \cup \{\#\}) \times \{0,1\}^k$ which is the lexicographically ordered sequence of every possible marking of k positions in $\#u$. For instance, we have $\mathcal{C}_1(ab) = \begin{matrix} \# & a & b & \# & a & b & \# & a & b \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{matrix}$ and

$$\mathcal{C}_2(ab) = \begin{matrix} \# & a & b & \# & a & b & \# & a & b & \# & a & b & \# & a & b & \# & a & b & \# & a & b \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{matrix}.$$

▶ **Lemma B.1.** *Given an integer $k \geq 1$ and an alphabet Σ, one can construct a reversible k-pebble transducer $\mathcal{C}_k$ with $\mathcal{O}(k)$ states such that for all input word $u \in \Sigma^*$, $\mathcal{C}_k(u)$ is the marking of u for k-configurations.*

Proof. The machine for $\mathcal{C}_1$ is given in Figure 6 (left). It is a slightly modified version of the squaring function of Figure 1, mainly to accomodate the fact that $\mathcal{C}_1$ also produces the endmarker # at each iteration.

For $k > 1$, the machine $\mathcal{C}_k$ is defined using an enhanced $\mathcal{C}_{k-1}$ as a blackbox. Indeed, the marking for k-configurations over a word u can be defined as the $k-1$ marking iterated $|\#u|$ times, where each iteration is marked by the new pebble on a different position of $\#u$, in order. Then $\mathcal{C}_k$ uses a modified $\mathcal{C}_{k-1}^{+1}$ that outputs its marking plus the one of the new pebble. It is described in Figure 6 (right). ◀

▶ **Lemma B.2.** *Given an integer $k \geq 1$ and an alphabet Σ, one can construct a reversible 0-pebble transducer $\mathcal{C}_k^{=}$ with $\mathcal{O}(2^{k^2})$ states that reads $[\![\mathcal{C}_k]\!](u)$ and adds to each copy of $\#u$ in $[\![\mathcal{C}_k]\!](u)$ a $k \times k$ boolean matrix M such that $M_{i,j} = 1$ if and only if i and j mark the same position in this copy of $\#u$.*

Sketch of proof. Informally, the transducer $\mathcal{C}_k^=$ reads a marking of $\#u$ and computes the matrix M along this copy. Then, when reaching a new pair $(\#, b)$, it goes back to output the copy enriched with the matrix M. Special care has to be taken to ensure reverse-determinism. In particular, the transducer $\mathcal{C}_k^=$ needs to undo the computation done before moving again to the next marked copy of $\#u$. Details can be found in [11]. ◀

▶ **Lemma B.3.** *Given a k-pebble ($k \geq 1$) transducer $\mathcal{T}$ with n states, one can construct a 0-pebble transducer $\mathcal{T}_0$ with $\mathcal{O}(kn)$ states such that $[\![\mathcal{T}]\!] = [\![\mathcal{T}_0]\!] \circ [\![\mathcal{C}_k^=]\!] \circ [\![\mathcal{C}_k]\!]$.*

Proof. The idea is to simulate the moves of the pebbles of $\mathcal{T}$ with moves of the reading head of $\mathcal{T}_0$ along the sequence of k-configurations produced by $\mathcal{C}_k$. The tests φ are validated using the information added by $\mathcal{C}_k^=$.

Informally, the transducer $\mathcal{T}_0$ remembers in its state the number of pebbles dropped by $\mathcal{T}$. It uses the configuration where the unused pebbles are on the same position as the reading head. Then, when the simulation of $\mathcal{T}$ drops a pebble i, $\mathcal{T}_0$ is already reading a configuration where the pebble i is at the current position. The transducer $\mathcal{T}_0$ simply needs to increment the number of pebbles dropped. Conversely, if a pebble i is lifted, it means the reading head is on a position where the pebble i is present. Hence it only needs to decrement its number of pebbles dropped.

The move of the reading head uses the fact that the configurations are output by $\mathcal{C}_k$ in lexicographic order. If the head moves to the left while having i pebbles dropped, $\mathcal{T}_0$ moves left until it sees a position where pebbles $i+1$ to k are all present. The first such position is the previous one from where we started due to the lexicographic order. It is also the correct position to maintain the invariant that all undropped pebbles are placed at the same position as the head. A move right is treated symmetrically.

Because the input word of u is read in a cyclic fashion, this might create issue for $\mathcal{T}_0$ as consecutive endmarkers $\#$ do not belong to the same marking. More precisely, given any position in $\mathcal{C}_k(u)$, its corresponding endmarker is the closest (possibly itself) $\#$-labelled position on its left. So if during the computation $\mathcal{T}_0$ reaches the endmarker $\#$ moving left and need to keep moving left, $\mathcal{T}_0$ needs to move to a position where all pebbles 1 to i are unchanged, but pebbles $i+1$ to k are on the last letter of the word. This is done by reaching the closest position on the right where pebbles $i+1$ to k are on an endmarker $\#$ (or $\vdash$ in the case where we reach the end of the word) then moving to the position on the left.

Symmetrically, if the computation of $\mathcal{T}_0$ reaches an endmarker $\#$ with a single right move, it means that pebbles $i+1$ to k were on the last position of u, then $\mathcal{T}_0$ moves left to the closest position where pebbles $i+1$ to k are all on the endmarker $\#$.

Formally, let $\mathcal{T} = (Q, \Sigma, \delta, k, q_i, q_f, \Gamma, \mu)$. We define $\mathcal{T}_0 = (P, \Sigma', \gamma, 0, p_i, p_f, \Gamma, \nu)$ where $\Sigma' = \Sigma \times \{0,1\}^k \times \{0,1\}^{k^2}$, $P \subseteq (Q \times \{0,\dots,k\} \times \{s, m_r, m_\ell\}) \cup \{p_i, p_f\}$ where s stands for simulation, and m_ℓ, m_r stand for move left and right respectively. We divide P into

$$
\begin{aligned}
P_0 &= (Q \times \{0,\dots,k\} \times \{s\}) \cup \{p_i, p_f\},\\
P_{+1} &= Q \times \{0,\dots,k\} \times \{m_r\},\\
P_{-1} &= Q \times \{0,\dots,k\} \times \{m_\ell\}\,.
\end{aligned}
$$

To avoid the more convoluted cases, we assume that in $\mathcal{T}$ transitions with action drop or lift do not move the reading head (this is especially needed for $\#$). Note that this can be enforced by decomposing a drop and move transition into two transitions (see Lemma 4.5).

For $0 \leq i \leq k$, we denote by b^{+i} any vector where $b_j^{+i} = 1$ for all $i < j \leq k$ and we denote by b^{-i} any vector where $b_j^{-i} \neq 1$ for some $j > i$.

Given an integer $0 \leq i \leq k$, a bit vector b and a matrix M, we say that $b, M, i \models (\mathsf{h} = \mathsf{p}_\ell)$ (resp. $\mathsf{p}_\ell = \mathsf{p}_{\ell'}$) if $\ell \leq i$ and $b_\ell = 1$ (resp. $\ell, \ell' \leq i$ and $M_{\ell,\ell'} = 1$).

The transitions in γ and the output function ν are defined below, where a denotes a letter different from $\#$, σ is any letter (possibly $\#$), b is any k-bit vector and M is a $k \times k$ boolean matrix. We omit the tests and operations in transitions as $\mathcal{T}_0$ is pebbleless. All transitions output ε except those of Item 3 below.

1. $(p_i, \vdash, (q_i, 0, m_r))$ and $((q_i, 0, m_r), (\#, b^{+0}, M_1), (q_i, 0, s))$.
2. $((q_f, 0, s), (\#, b^{+0}, M_1), (q_f, 0, m_\ell))$ and $((q_i, 0, m_r), \vdash, p_f)$.
3. $((q, i, s), (\sigma, b^{+i}, M), (q', i', m))$ if there exists a transition $t = (q, \sigma, \varphi, \mathsf{op}, q')$ in $\mathcal{T}$ such that $b^{+i}, M, i \models \varphi$ and $\mathsf{op} \in \{\mathsf{nop}, \mathsf{drop}_{i+1}, \mathsf{lift}_i\}$ and $b_i = 1$ if $\mathsf{op} = \mathsf{lift}_i$. In this case,

$$i' = \begin{cases} i & \text{if } \mathsf{op} = \mathsf{nop} \\ i+1 & \text{if } \mathsf{op} = \mathsf{drop}_{i+1} \\ i-1 & \text{if } \mathsf{op} = \mathsf{drop}_i \end{cases} \quad \text{and} \quad m = \begin{cases} s & \text{if } q' \in Q_0 \\ m_\ell & \text{if } q' \in Q_{-1} \wedge \sigma \neq \# \\ m_r & \text{if } q' \in Q_{+1} \text{ or } q' \in Q_{-1} \wedge \sigma = \# \end{cases}$$

   The production of this transition is the same as t.
   Note that, the head of $\mathcal{T}_0$ moves left or right when $q' \in Q_{-1}$ or $q' \in Q_{+1}$. But in general, the head of $\mathcal{T}_0$ does not reach immediately the position where the simulation of $\mathcal{T}$ continues. We have to skip positions with a bit vector of the form $b'^{-i'}$ and handle carefully the endmarker $\#$. This is the purpose of the following transitions.
4. If $q \in Q_{-1}$ then $((q, i, m_\ell), (\sigma, b^{-i}, M), (q, i, m_\ell))$ and $((q, i, m_\ell), (\sigma, b^{+i}, M), (q, i, s))$.
5. If $q \in Q_{-1}$ then $((q, i, m_r), (a, b, M), (q, i, m_r))$ and $((q, i, m_r), (\#, b^{-i}, M), (q, i, m_r))$ and $((q, i, m_r), (\#, b^{+i}, M), (q, i, m_\ell))$ and $((q, i, m_r), \vdash, (q, i, m_\ell))$.
   Note that, when moving right we reach $\#$ with a bit vector of the form b^{+i} then the previous letter also has a bit vector of the form b^{+i}. Hence, the transition taken from (q, i, m_ℓ) will go directly to the simulation mode (q, i, s). If moving right we reach the endmarker $\vdash$ then the previous letter has a bit vector of the form b^{+0}.
6. If $q \in Q_{+1}$ then $((q, i, m_r), (\sigma, b^{-i}, M), (q, i, m_r))$ and $((q, i, m_r), (a, b^{+i}, M), (q, i, s))$.
7. If $q \in Q_{+1}$ then $((q, i, m_r), (\#, b^{+i}, M), (q, i, m_\ell))$ and $((q, i, m_r), \vdash, (q, i, m_\ell))$ and $((q, i, m_\ell), (a, b, M), (q, i, m_\ell))$ and $((q, i, m_\ell), (\#, b^{-i}, M), (q, i, m_\ell))$ and $((q, i, m_\ell), (\#, b^{+i}, M), (q, i, s))$.

We remark that $\mathcal{T}_0$ is obtained from $\mathcal{T}$ by decomposing its transitions into separate sequences of transitions. The bit vectors and the matrices M allow us to check the tests of transitions of $\mathcal{T}$. The producing transitions of $\mathcal{T}_0$ correspond to the ones of $\mathcal{T}$ and the accepting runs of $\mathcal{T}_0$ correspond to the accepting runs of $\mathcal{T}$. Hence for any input word u, $u \in \mathsf{dom}(\mathcal{T})$ if, and only if $[\![\mathcal{C}_k^=]\!]([\![\mathcal{C}_k]\!](u)) \in \mathsf{dom}(\mathcal{T}_0)$ and $[\![\mathcal{T}]\!] = [\![\mathcal{T}_0]\!] \circ [\![\mathcal{C}_k^=]\!] \circ [\![\mathcal{C}_k]\!]$. ◀

On the Send-Synchronizability Problem for Mailbox Communication

Romain Delpy
LaBRI, Univ. Bordeaux, CNRS, Bordeaux INP, Talence, France

Anca Muscholl
LaBRI, Univ. Bordeaux, CNRS, Bordeaux INP, Talence, France

Grégoire Sutre
LaBRI, Univ. Bordeaux, CNRS, Bordeaux INP, Talence, France

Abstract

A system of communicating automata is send-synchronizable if its set of send sequences (i.e., the projection on send actions of its executions) is the same when communications are asynchronous and when they are rendez-vous synchronizations. Send-synchronizability was claimed to be decidable for the mailbox semantics (Basu and Bultan, 2011) and for the peer-to-peer semantics (Basu and Bultan, 2016). Finkel and Lozes showed in 2017 that the proofs of these results are flawed, and they proved that send-synchronizability is in fact undecidable for peer-to-peer systems. The send-synchronizability problem for mailbox systems was left open. A partial solution was recently proposed in (Di Giusto, Laversa and Peters, 2024). In this paper, we revisit the send-synchronizability problem for mailbox systems. Firstly, we show that send-synchronizability is undecidable for mailbox systems, thus closing the question left open in (Finkel and Lozes, 2023) and (Di Giusto, Laversa and Peters, 2024). Secondly, we show that send-synchronizability is decidable for the class of 1-schedulable mailbox systems. A system is 1-schedulable if every execution can be re-scheduled into an equivalent execution where each send is either immediately followed by its matching receive, or is never matched. Despite the apparent similarity between send-synchronizability and 1-schedulability, the proof that send-synchronizability is decidable for 1-schedulable mailbox systems is quite involved. We believe that the techniques that we develop in this proof could be used to address other problems on mailbox systems, such as the realizability problem.

2012 ACM Subject Classification Theory of computation → Logic and verification

Keywords and phrases Concurrent programming, Mailbox communication, Verification, Synchronizability

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2025.15

Funding This work was (partially) supported by the grant ANR-23-CE48-0005 of the French National Research Agency ANR (project PaVeDyS).

1 Introduction

Message-passing is a key synchronization mechanism for concurrent programming and distributed systems. In this model, processes running asynchronously synchronize by exchanging messages over unbounded channels. Typically, the communication follows a peer-to-peer model, which is particularly useful for reasoning about telecommunication protocols.

Recently, mailbox communication has attracted increased attention due to its role in multi-thread programming, as seen in languages like Rust or Erlang. In mailbox communication each process has a single incoming communication buffer, or mailbox, where messages from other processes are multiplexed.

© Romain Delpy, Anca Muscholl, and Grégoire Sutre;
licensed under Creative Commons License CC-BY 4.0

Asynchronous communication poses significant challenges for formal verification, since it can easily emulate Turing machines. The quest for conditions that ensure decidability for automatic verification has led to various restrictions, such as constraints on channel behavior (e.g. lossiness [1, 12], or properties based on partial-orders [18, 14, 20]). More recently, research has focused on restricting the communication protocols themselves [6, 16, 17, 8].

While synchronous communication offers great simplicity, its relevance extends beyond verification. Many specification formalisms, such as choreography languages [2] and multi-party session types [24, 22, 23], are built upon synchronous communication. This raises a fundamental question, considering the inherent asynchrony of real-world systems: how different is an asynchronous system from its idealized synchronous version, where every sent message is instantaneously received?

Send-synchronizability offers one way to compare synchronous and asynchronous semantics. The work [13] defined a system as (send-)synchronizable if the sequence of sent messages is identical under both asynchronous and synchronous semantics. While [2] claimed decidability for mailbox systems and later extended the claim to peer-to-peer systems [3], the latter claim was refuted by [10, 11], showing undecidability for peer-to-peer systems. The decidability of send-synchronizability for mailbox systems, however, remained an open question (see, e.g., [9]), and this paper addresses it.

Send-synchronizability has been explored in several subsequent works [2, 4, 3, 11, 9] with variations such as final states, or requiring that the same state is reached in both the synchronous and asynchronous execution. For example, in [9], each process has a set of final states. This generalized version was shown to be undecidable in [9], with final states playing a key role in the proof. Our first contribution is to show that send-synchronizability remains undecidable even in the standard setting of communicating systems without final states.

Our second contribution shows the decidability of send-synchronizability for 1-schedulable systems. These systems, termed 1-synchronizable in [6] but renamed here for clarity, share similarities with the synchronous semantics. However, a key difference is that not all messages need to be received. A system is 1-schedulable if every execution can be reordered into an equivalent one where each send is either immediately followed by its corresponding receive, or remains unmatched. Being 1-schedulable is a decidable property, with PSPACE complexity [6, 16]. We show that send-synchronizability is decidable for 1-schedulable systems, but the proof is surprisingly intricated because of the unreceived messages. Our proof employs partial-order techniques (commutations), with the main challenge being the unmatched send events.

For convenience, technical terms and notations in the electronic version of this manuscript are hyper-linked to their definitions (cf. `https://ctan.org/pkg/knowledge`).

2 Definitions and Notations

2.1 Mailbox message-passing systems

We let $\mathbb{P}$ denote a finite non-empty set of *processes*, and $\mathbb{M}$ denote a finite non-empty set of *message contents*. The set of (communication) *actions* is $Act = \{p!q(m), q?p(m) \mid p, q \in \mathbb{P}, p \neq q, m \in \mathbb{M}\}$. An action $p!q(m)$ denotes a *send* by p of message m to q and an action $p?q(m)$ denotes a *receive* by p of message m from q. In both cases, the process performing the action is p. A communicating finite-state machine [7] is a finite set of processes that exchange messages, each process being given as a finite LTS. Recall that a (finite) *labeled transition system*, *LTS* for short, is a quadruple $(L, A, \to, i)$ where L is a (finite) set of *states*, A is a finite alphabet, $\to \subseteq L \times A \times L$ is a set of *transitions*, and $i \in L$ is an *initial* state. In the following definition, Act_p denotes the set of actions $a \in Act$ performed by p.

A *Communicating Finite-State Machine* (CFM for short) is a tuple $\mathcal{A} = (\mathcal{A}_p)_{p\in\mathbb{P}}$, where each $\mathcal{A}_p$ is a finite LTS $\mathcal{A}_p = (L_p, Act_p, \to_p, i_p)$. States in L_p are called *local states*. The *size* of $\mathcal{A}$ is defined as $\sum_{p\in\mathbb{P}}(|L_p| + |\to_p|)$.

The usual semantics of communication is peer-to-peer, with (at most one) fifo channel for each pair of distinct processes. In this paper we focus on the *mailbox semantics*. Here, every process has a receiving fifo queue in which every other process can enqueue messages. We define the semantics by an associated global transition system.

Let $\mathcal{A} = (\mathcal{A}_p)_{p\in\mathbb{P}}$ be a CFM. The *global transition system* associated with $\mathcal{A}$ is the LTS $\mathcal{T}(\mathcal{A}) = (C_\mathcal{A}, Act, \to_\mathcal{A}, c_{in})$ with set of configurations $C_\mathcal{A} = G \times \prod_{p\in\mathbb{P}}(\mathbb{P}\times\mathbb{M})^*$ consisting of global states $G = \prod_{p\in\mathbb{P}} L_p$ (i.e., products of local states) and queue contents over the alphabet $\mathbb{P}\times\mathbb{M}$. Let $((\ell_p)_{p\in\mathbb{P}}, (w_p)_{p\in\mathbb{P}}) \xrightarrow{a}_\mathcal{A} ((\ell'_p)_{p\in\mathbb{P}}, (w'_p)_{p\in\mathbb{P}})$ if

- $\ell_p \xrightarrow{a}_p \ell'_p$ and $\ell_q = \ell'_q$ for $q \neq p$, where p is the process performing a.
- Send actions: if $a = p!q(m)$ then $w'_q = w_q\,(p,m)$ and $w'_{p'} = w_{p'}$ for $p' \neq q$.
- Receive actions: if $a = p?q(m)$ then $(q,m)\,w'_p = w_p$ and $w'_{p'} = w_{p'}$ for $p' \neq p$.

The initial configuration is $c_{in} = ((i_p)_{p\in\mathbb{P}}, \varepsilon^\mathbb{P})$.

An *execution* of $\mathcal{T}(\mathcal{A})$ is a sequence $\rho = c_0 \xrightarrow{a_1} c_1 \cdots \xrightarrow{a_n} c_n$ with $c_i \in C_\mathcal{A}$ such that $c_{i-1} \xrightarrow{a_i}_\mathcal{A} c_i$ for every i. The sequence $a_1 \cdots a_n$ is the *label* of the execution. The execution is *initial* if $c_0 = c_{in}$.

▶ Remark 2.1. Note that the queue content includes the identity of the sender. This is to exclude executions labelled by $p!q(m)\ q?r(m)$ with $p \neq r$. Without this addition such executions would be allowed in the mailbox semantics, which is clearly not intended.

▶ **Definition 2.2** (Trace). *A* trace *of a CFM $\mathcal{A}$ is a sequence $u \in Act^*$ such that there exists an initial execution of $\mathcal{T}(\mathcal{A})$ labelled by u. The set of all traces of $\mathcal{A}$ is denoted by $Tr(\mathcal{A})$.*

▶ **Definition 2.3** (Viable). *A sequence $v \in Act^*$ is called* viable *if for every $p \in \mathbb{P}$:*

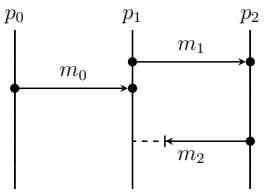
- *for every prefix u of v, the number of receives of p in u is less or equal the number of sends to p in u;*
- *for every k, if the k-th receive of p in v has label $p?q(m)$ then the k-th send to p in v has label $q!p(m)$.*

▶ **Definition 2.4** (Rendez-vous). *A sequence from Act^* is called a* rendez-vous *sequence if it is of the form $p_1!q_1(m_1)q_1?p_1(m_1)\ldots p_n!q_n(m_n)q_n?p_n(m_n)$. Note that every send action is matched in a rendez-vous sequence, and that such sequences are viable. We define $Tr_{rdv}(\mathcal{A}) \subseteq Tr(\mathcal{A})$ as the set of traces in $Tr(\mathcal{A})$ that are rendez-vous.*

The classical *happens-before* relation [21], frequently used in reasoning about distributed systems, orders the actions of each process and every (matched) send action before its matching receive. Because of mailbox semantics, an additional order arise between sends to the same process. The happens-before relation together with the mailbox order naturally associates a partial order with every trace, known as *message sequence chart*:

▶ **Definition 2.5** (Message Sequence Chart). *An* MSC *over $\mathbb{P}$ is an Act-labeled partially ordered set $\mathcal{M} = (E, \preceq, \lambda)$ of events E, with $\lambda : E \to Act$ and $\preceq = (\leq_\mathbb{P} \cup \mathrm{msg} \cup <_{mb})^*$ the least partial order containing the relations $\leq_\mathbb{P}$, msg and $<_{mb}$, which are defined as:*

1. *For every process p, the set of events on p is totally ordered by $\leq_\mathbb{P}$, and $\leq_\mathbb{P}$ is the union of these total orders.*
2. msg *is the set of matching send/receive event pairs. In particular, $(e,f) \in$ msg implies $\lambda(e) = p!q(m)$ and $\lambda(f) = q?p(m)$ for some $p, q \in \mathbb{P}$ and $m \in \mathbb{M}$. Moreover,* msg *is a partial bijection between sends and receives such that every receive is paired with a (unique) send. A send is called* matched *if it is in the domain of* msg*, and* unmatched *otherwise.*



$p_0!p_1(m_0)\, p_1!p_2(m_1)\, p_2?p_1(m_1)\, p_2!p_1(m_2)\, p_1?p_0(m_0)$

Figure 1 A sequence and its MSC. An unmatched send action is marked by a special arrowhead, as for m_2. We have here $s_0 <_{\mathtt{mb}} s_2$, so $s_0 \prec s_2$, and $s_1 \parallel s_0$, with s_i the send of message m_i.

3. *for every process p, we order two sends $e <_{mb} e'$ with $e = q!p(m)$ and $e' = q'!p(m')$ if*
   - *either e is matched and e' is unmatched,*
   - *or $(e, f), (e', f') \in \mathrm{msg}$ and $f <_{\mathbb{P}} f'$.*

   We call $<_{mb}$ the mailbox order.

For two events $e, f \in E$ we write $e \parallel f$ if neither $e \preceq f$ nor $f \preceq e$ holds. For a set of events $F \subseteq E$ we write $Past(F) = \{f \mid f \preceq e \text{ for some } e \in F\}$ and $Ftr(F) = \{f \mid e \preceq f \text{ for some } e \in F\}$ for the causal past and future of F, respectively.

If $u = u[1] \dots u[n]$ is a viable sequence of actions, then we can associate an MSC with u by setting $\mathtt{msc}(u) = (E, \preceq, \lambda)$ with $E = \{e_1, \dots, e_n\}$, $\lambda(e_i) = u[i]$, and the orders defined as expected:

- $e_i \leq_{\mathbb{P}} e_j$ if $u[i]$ and $u[j]$ are performed by the same process and $i \leq j$.
- $(e_i, e_j) \in \mathrm{msg}$ if there exists $k \geq 1$ and a process p such that $u[i]$ is the k-th send to p and $u[j]$ is the k-th receive of p.
- $e_i <_{\mathtt{mb}} e_j$ if $u[i]$ and $u[j]$ are both sends to the same process p, $u[i]$ is matched to some $u[k]$, and either $u[j]$ is unmatched, or $u[j]$ is matched to some $u[\ell]$ with $k \leq \ell$.

▶ Remark 2.6. By definition, for any viable sequence u, the associated MSC $\mathtt{msc}(u)$ is well-defined. For the converse, for every MSC $\mathcal{M}$, every (labelled) linearization of the partial order $\preceq$ of $\mathcal{M}$ is viable. Indeed, all receives of the same process are totally ordered by $\leq_{\mathbb{P}}$, and the corresponding sends are ordered in the same way because of the mailbox order. For example, the sequence shown in Figure 1 is viable, but $p_1!p_2(m_1)\, p_2?p_1(m_1)\, p_2!p_1(m_2)\, p_0!p_1(m_0)\, p_1?p_0(m_0)$ is not.

The next definition introduces an equivalence relation $\equiv$ on sequences of actions. Two viable sequences are equivalent up to commuting adjacent actions that are not ordered by being on the same process, being a matching send/receive pair, or being two sends to the same process that are not both unmatched.

▶ **Definition 2.7** (Equivalence $\equiv$). *Two viable sequences $u, v \in Act^*$ are called* equivalent *if $\mathtt{msc}(u) = \mathtt{msc}(v)$ (up to isomorphism), and we write $u \equiv v$ in this case.*

In this paper, we define (as in [2, 10]) the *observable behavior* of a CFM as the set of projections on sends. We set $S = \{p!q(m) \mid p, q \in \mathbb{P}, p \neq q, m \in \mathbb{M}\}$. For a sequence $u \in Act^*$, we denote by $u|_S$ the projection of u on S. For any $X \subseteq Act^*$, we write $X|_S = \{u|_S \mid u \in X\}$.

▶ **Definition 2.8** (Send-synchronizability). *Let $\mathcal{A}$ be a CFM. We say that $\mathcal{A}$ is* send-synchronizable *if $Tr(\mathcal{A})|_S = Tr_{rdv}(\mathcal{A})|_S$.*

The *send-synchronizability problem* is the question whether a CFM is send-synchronizable.

3 Undecidability of Send-Synchronizability for Mailbox Systems

We show in this section that the send-synchronizability problem is undecidable. We rely in our undecidability proof on the following decision problem, which is a variant of the well-known Post correspondence problem.

PRE-MPCP	
Input:	A finite sequence of pairs $(x_1, y_1), \ldots, (x_K, y_K)$, with $K \geq 1$, where $x_i, y_i \in \Sigma^*$ for some finite alphabet Σ.
Output:	Yes if there exists a sequence of indices $i_1, \ldots, i_k \in \{1, \ldots, K\}$, with $k \geq 1$, such that $i_1 = 1$, $x_{i_1} \cdots x_{i_k} = y_{i_1} \cdots y_{i_k}$ and $\lvert x_{i_1} \cdots x_{i_j} \rvert \geq \lvert y_{i_1} \cdots y_{i_j} \rvert$ for every $j \in \{1, \ldots, k\}$.

It is well-known that PRE-MPCP is undecidable [19]. The remainder of this section provides a reduction from PRE-MPCP to send-synchronizability.

Consider an instance $\mathcal{I} = (x_1, y_1), \ldots, (x_K, y_K)$ of PRE-MPCP over a finite alphabet Σ. We construct a CFM $\mathcal{A}(\mathcal{I})$ such that $\mathcal{I}$ is a positive instance of PRE-MPCP if, and only if, $\mathcal{A}(\mathcal{I})$ is not send-synchronizable. The CFM $\mathcal{A}(\mathcal{I})$ comprises three processes, namely G (the guesser), R (the relayer), and V (the verifier). So $\mathbb{P} = \{G, R, V\}$. The set of message contents is $\mathbb{M} = \Sigma \cup \overline{\Sigma} \cup \{\$, \overline{\$}, \#\}$, where $\overline{\Sigma} = \{\overline{a} \mid a \in \Sigma\}$ is disjoint from Σ, and the messages $\$$, $\overline{\$}$ and $\#$ denote three distinct symbols that are not in $\Sigma \cup \overline{\Sigma}$. We choose to introduce a copy $\overline{\Sigma} \cup \{\overline{\$}\}$ of the alphabet $\Sigma \cup \{\$\}$ in order to easily[1] distinguish in the mailbox of V the messages that were sent by G from the messages that were sent by R. The symbols $\$$ and $\overline{\$}$ are used to mark the end of the sequences $x_{i_1} \cdots x_{i_k}$ and $\overline{y_{i_1}} \cdots \overline{y_{i_k}}$ guessed by G. It is understood that $\overline{y_1}, \ldots, \overline{y_K}$ are the words in $\overline{\Sigma}$ that are obtained from $y_1, \ldots, y_K$ by replacing each letter a by $\overline{a}$. The message $\#$ is sent by V when the two guessed sequences are equal. The LTSes $\mathcal{A}_G$, $\mathcal{A}_R$ and $\mathcal{A}_V$ of the processes G, R and V are defined in Figures 2–4. To ease understanding, the resulting communication topology, along with the potential message contents in the channels, is depicted in Figure 5. In the description of the guesser process (see Figure 2), we use the shortcuts $G!R(x_i)$ and $G!V(\overline{y_i})$ to reduce clutter. Let us explain what we mean by these shortcuts. A transition $\ell \xrightarrow{p!q(m_1 \cdots m_n)} \ell'$, where $m_i \in \mathbb{M}$, stands for the contiguous sequence of transitions $\ell = \ell_0 \xrightarrow{p!q(m_1)} \ell_1 \cdots \ell_{n-1} \xrightarrow{p!q(m_n)} \ell_n = \ell'$ where $\ell_1, \ldots, \ell_{n-1}$ are intermediate states. The observant reader will notice that the LTSes $\mathcal{A}_R$ and $\mathcal{A}_V$ contain silent transitions (from the initial state to the states D and N), which are not allowed in our definition of CFM. This is merely a notational convenience to reduce clutter in the figures. We may safely get rid of these silent transitions since send-synchronizability of a CFM is not sensitive to such transitions.

The behavior of the guesser process G should be quite clear. It sends the two guessed sequences and then waits for a message $\#$ from V, which it relays to R, signaling that the guessed sequences form a solution. The relayer process R has two operating modes, a *normal* mode (right-hand side of Figure 3) and a *dummy* mode (left-hand side of Figure 3). In the normal mode, R simply relays to V every message that it receives from G. For V to process its mailbox properly (see below), the send actions $G!V(\cdot)$ and $R!V(\cdot)$ need to be

[1] In our global transition system $\mathcal{T}(\mathcal{A})$, each message content stored in a buffer is decorated with the channel of the corresponding send action. So our construction would also work without the copy $\overline{\Sigma} \cup \{\overline{\$}\}$ of $\Sigma \cup \{\$\}$, but we believe that this copy makes our construction easier to read.

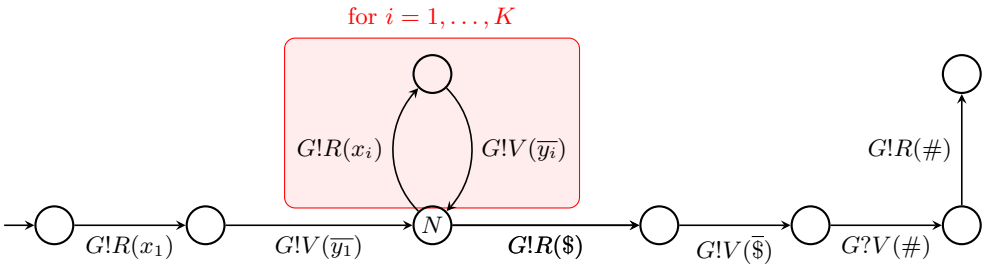


Figure 2 LTS $\mathcal{A}_G$ of the guesser process G of the CFM $\mathcal{A}(\mathcal{I})$.

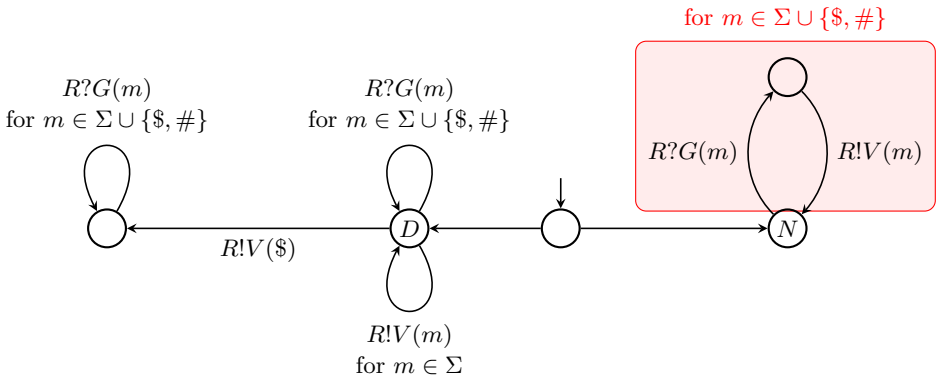


Figure 3 LTS $\mathcal{A}_R$ of the relayer process R of the CFM $\mathcal{A}(\mathcal{I})$.

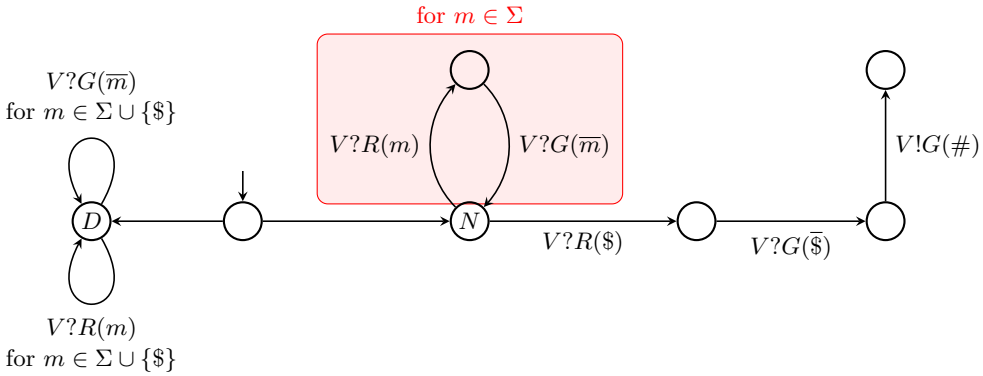


Figure 4 LTS $\mathcal{A}_V$ of the verifier process V of the CFM $\mathcal{A}(\mathcal{I})$.

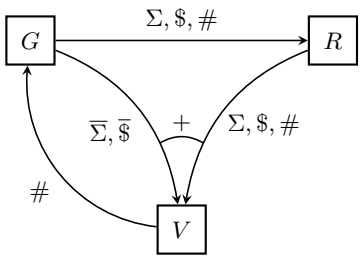


Figure 5 Topology of the CFM $\mathcal{A}(\mathcal{I})$ and message contents potentially exchanged along the channels. The $+$ sign serves as a reminder that the channels (G, V) and (R, V) are multiplexed into a single mailbox.

interleaved in lock-step (but there may be executions in $\mathcal{T}(\mathcal{A}(\mathcal{I}))$ where this does not hold). The purpose of the dummy mode is to make $\mathcal{A}(\mathcal{I})$ send-synchronizable if $\mathcal{I}$ is a negative instance of Pre-MPCP. Notice that there is no switch between modes. Analogously, the verifier process V has a *normal* mode and a *dummy* mode (right-hand and left-hand sides of Figure 4, respectively). In the normal mode, V checks that the contents of its mailbox is of the form $a_1\overline{a_1}\cdots a_n\overline{a_n}\$\overline{\$}$. It then sends $\#$ to G if that is the case (otherwise it is blocked). The purpose of the dummy mode is the same as for R. Observe that the message $\#$ sent by V to G is relayed by G to R, and then relayed by R back to V, but V never receives it.

We first show that if $\mathcal{I}$ is a positive instance of Pre-MPCP, then $\mathcal{A}(\mathcal{I})$ is not send-synchronizable.

▶ **Lemma 3.1.** *If $\mathcal{I}$ is a positive instance of Pre-MPCP then there is an initial execution in $\mathcal{T}(\mathcal{A}(\mathcal{I}))$ leading to the configuration $(N, N, N; \varepsilon, \varepsilon, \varepsilon)$.*

Proof sketch. G guesses a solution $i_1, \ldots, i_k$, and sends $x_{i_1}\cdots x_{i_k}\$$ and $\overline{y_{i_1}}\cdots\overline{y_{i_k}}\overline{\$}$ to R and V, respectively. R relays $x_{i_1}\cdots x_{i_k}\$$ to V with the appropriate delay so that the mailbox of V is filled by G and R in lock-step. This is possible because our variant of PCP requires that $|x_{i_1}\cdots x_{i_j}| \geq |y_{i_1}\cdots y_{i_j}|$ for every $j \in \{1, \ldots, k\}$. We end up with $a_1\overline{a_1}\cdots a_n\overline{a_n}\$\overline{\$}$ in the mailbox of V. Then V empties its mailbox by iterating the cycle on N. ◀

▶ **Corollary 3.2.** *If $\mathcal{I}$ is a positive instance of Pre-MPCP then there is a trace $u \in Tr(\mathcal{A}(\mathcal{I}))$ containing the action $R!V(\#)$, hence, $\mathcal{A}(\mathcal{I})$ is not send-synchronizable.*

Now we show that if $\mathcal{I}$ is a negative instance of Pre-MPCP, then $\mathcal{A}(\mathcal{I})$ is send-synchronizable. Some additional notations are needed. For every process $p \in \mathbb{P}$, we let $L(\mathcal{A}_p)$ denote the language recognized by the LTS $\mathcal{A}_p$ viewed as a non-deterministic automaton with every state final, and we let $u_{|p}$ denote the projection of a sequence $u \in Act^*$ on Act_p. We also define $match : S^* \to Act^*$ as the morphism given by $match(p!q(m)) = p!q(m)\, q?p(m)$.

▶ **Lemma 3.3.** *If there is a trace in $Tr(\mathcal{A}(\mathcal{I}))$ containing the action $R!V(\#)$ then $\mathcal{I}$ is a positive instance of Pre-MPCP.*

▶ **Lemma 3.4.** *For every trace $u \in Tr(\mathcal{A}(\mathcal{I}))$, if u does not contain the action $R!V(\#)$ then $match(u|_S)_{|p} \in L(\mathcal{A}_p)$ for every process $p \in \{G, R, V\}$.*

Proof sketch. Either u contains no send action to G, or u contains a single send action to G, which is $V!G(\#)$. In that case, this action is necessarily after $G!V(\overline{\$})$ and before $G!R(\#)$. In both cases, we get that $match(u|_S)_{|G} \in L(\mathcal{A}_G)$.

All sends to R are from G and all sends by R are to V. The word sent by G to R in u is a prefix of a word in $\Sigma^*\$\#$. The word sent by R to V in u is a prefix of a word in $\Sigma^*\$$, since u does not contain the action $R!V(\#)$. It follows that $match(u|_S)_{|R} \in L(\mathcal{A}_R)$, by letting R select the dummy mode.

The word sent to V in u is in $(\Sigma \cup \overline{\Sigma} \cup \{\$, \overline{\$}\})^*$, since u does not contain the action $R!V(\#)$. If u contains no send action by V, then $match(u|_S)_{|V} \in L(\mathcal{A}_V)$, by letting V select the dummy mode. Otherwise, u contains a single send action by V, which is $V!G(\#)$. The definition of $\mathcal{A}_V$ (together with the mailbox semantics) entails that the word sent to V in u is of the form $a_1\overline{a_1}\cdots a_n\overline{a_n}\$\overline{\$}$. Moreover, the action $V!G(\#)$ is necessarily after $G!V(\overline{\$})$. We derive that $match(u|_S)_{|V}$ coincides with the sequence of actions by V in u, hence, $match(u|_S)_{|V} \in L(\mathcal{A}_V)$. ◀

▶ **Corollary 3.5.** *If $\mathcal{I}$ is a negative instance of* Pre-MPCP *then $\mathcal{A}(\mathcal{I})$ is send-synchronizable.*

We then get the next theorem from Corollaries 3.2 and 3.5.

▶ **Theorem 3.6.** *The send-synchronizability problem is undecidable.*

4 Send-synchronizability and 1-schedulable Systems

In this section we show that we can decide send-synchronizability for a subclass of CFMs that we call 1-schedulable.[2] We start by defining 1-schedulable systems and a commutation relation over send sequences. Then we show that if all traces of a 1-schedulable CFM respect a certain property (and we say the CFM is *good* in this case) then we can check if the CFM is send-synchronizable. Next we show that any 1-schedulable CFM that is not good, cannot be send-synchronizable. Finally we show that we can check if a 1-schedulable CFM is good.

▶ **Definition 4.1** (1-scheduling and 1-schedulable)**.** *A viable sequence $u \in Act^*$ is a* 1-scheduling *if every send action in u is either directly followed by its receive or it is unmatched. We also say in that case that u is* 1-scheduled*. A sequence $u \in Act^*$ is* 1-schedulable *if there exists $v \equiv u$ such that v is a 1-scheduling.*

A CFM $\mathcal{A}$ is 1-schedulable if every trace $u \in Tr(\mathcal{A})$ is 1-schedulable.

From [8] (see also [16]) we know that 1-schedulability is a decidable property (PSPACE-complete). To make the characterization formal, we need some notation first. Given a binary relation $\bowtie$ over some set X, a $\bowtie$-cycle is any sequence $(x_1, \ldots, x_k) \in X$, with $k \geq 1$, such that $x_1 \bowtie x_2 \cdots \bowtie x_{k+1}$, where $x_{k+1} := x_1$. A viable sequence w is not 1-schedulable if, and only if, $\mathtt{msc}(w)$ contains a $(\prec \cup \mathrm{msg}^{-1})$-cycle with at least two distinct sends [8]. For example, the sequence $1!2(m)\, 2!1(m')\, 1?2(m')\, 2?1(m)$ is not 1-schedulable, as witnessed by the cycle $1!2(m) <_{\mathbb{P}} 1?2(m')\ \mathrm{msg}^{-1}\ 2!1(m') <_{\mathbb{P}} 2?1(m)\ \mathrm{msg}^{-1}\ 1!2(m)$.

Throughout this section we will assume that the CFM $\mathcal{A}$ is 1-schedulable. Our characterization of send-synchronizability will rely on partial commutations applied to the observable behaviors of rendez-vous traces.

▶ **Definition 4.2** (Commutations)**.** *We define a commutation relation $SI \subseteq S \times S$ on the alphabet S of sends as follows. Let $a = p!q(m)$ and $b = p'!q'(m')$ be two send actions. Then $(a, b) \in SI$ if $p \neq p'$, $q \neq q'$ and $q \neq p'$.*

If $(a, b) \in SI$ and $w, w' \in S^*$ then we write $wabw' \Rightarrow_{SI} wbaw'$, and we consider the reflexive-transitive closure $\overset{*}{\Rightarrow}_{SI}$ of this relation. We denote the *commutative closure* of a send sequence $v \in S^*$ as $Cl_{SI}(v) = \{v' \in S^* \mid v \overset{*}{\Rightarrow}_{SI} v'\}$. We define $Cl_{SI}(E)$, for $E \subseteq S^*$, as $\bigcup_{v \in E} Cl_{SI}(v)$.

The example in Figure 6 gives a first intuition about the role of commutations for send-synchronizability. As seen there, if a CFM has an execution w as depicted on the left, and is send-synchronizable, then it has to include also an execution as depicted on the right. This entails that if $1!0(m)2!1(m')$ is an observable behavior, then so is $2!1(m')1!0(m)$.

▶ Remark 4.3. It is readily seen that one can check if a regular language L satisfies $L = Cl_{SI}(L)$. For this it is enough to check that $wabw' \in L$ entails $wbaw' \in L$, for all words w, w' and $(a, b) \in SI$. Such a test can be done in Pspace, if L is described by an automaton.

[2] This decidable class of systems originates from [6] and it is the same as *1-synchronizable* there. To better distinguish it from send-synchronizability we decided to rename it here into 1-schedulable. Comparison with other close notions can be found in Section A.

Figure 6 In the left scenario, the two send events s and s' satisfy $s \parallel s'$. Note that $(1!0(m), 2!1(m')) \in SI$, so $1!0(m)\,2!1(m') \Rightarrow_{SI} 2!1(m')\,1!0(m)$. To be send-synchronizable, a CFM must also allow the scenario on the right, where $s' \prec s$.

4.1 From 1-schedulings to arbitrary traces via commutations

In this subsection we start with a sufficient condition that ensures that the send projections of traces of $\mathcal{A}$ can be obtained via commutations from the send projections of 1-schedulings of $\mathcal{A}$ (see Proposition 4.5). The definitions that follow are used in Lemma 4.4, that states the relation between commutations and projections for a single trace.

Consider a viable sequence w. We introduce two binary relations $\dashrightarrow$ and $\ll^w_{\text{us}}$ over the set of events E of $\mathtt{msc}(w)$, defined as follows:

- $f \dashrightarrow e$ if there exists $p \in \mathbb{P}$ such that e is an unmatched send to p, f is a send from p, and $e \parallel f$.
- $e \ll^w_{\text{us}} f$ if there exists $p \in \mathbb{P}$ such that e and f are unmatched sends to p, e is before f in w, and $e \parallel f$.

We say that w has a *backward* $\dashrightarrow$ *arc* if there exists two events e, f such that $f \dashrightarrow e$ and e occurs before f in w. Consider for example the trace w_1 in Figure 7a left: we have $s_0 \dashrightarrow s_1 \dashrightarrow s_2 \dashrightarrow s_0$, so w_1 has a backward $\dashrightarrow$ arc.

The objective of the $\ll^w_{\text{us}}$ relation is to enforce the order of unmatched sends to the same process. Given two viable sequences w and w', we write $w \equiv_{\text{us}} w'$ if $w \equiv w'$ and for every process p, the order of unmatched sends to p is the same in w and w'. Notice that for every viable sequences w, w' such that $w \equiv w'$, we have $w \equiv_{\text{us}} w'$ if, and only if, the binary relations $\ll^w_{\text{us}}$ and $\ll^{w'}_{\text{us}}$ coincide.

▶ **Lemma 4.4.** *Let w be a 1-scheduling and suppose that w has no backward $\dashrightarrow$ arc. Then $Cl_{SI}(w|_S) = \{w'|_S \mid w'$ is a viable sequence and $w' \equiv_{\text{us}} w\}$.*

Proof. We write $\mathcal{W} = \{w'|_S \mid w' \equiv_{\text{us}} w\}$.

We first show $Cl_{SI}(w|_S) \subseteq \mathcal{W}$ recursively on the number of commutation steps. Initially $w|_S \in \mathcal{W}$. Now we take $u \in Cl_{SI}(w|_S) \cap \mathcal{W}$, and let $u = u_0\,x\,y\,u_1$ with $x, y \in S$, $u_0, u_1 \in S^*$ and $(x, y) \in SI$. Let $u' = u_0\,y\,x\,u_1$ be the sequence of sends obtained by commuting x and y. As $u \in \mathcal{W}$, there is some $v \equiv_{\text{us}} w$ such that $v|_S = u$. We suppose w.l.o.g. that $v = v_0\,x\,y\,v_1$: if there were receives between x and y, each one could either swap with x to the left, or with y to the right, since x, y cannot be on the same process, and the sequence obtained would be $\equiv_{\text{us}}$ equivalent to v. As $(x, y) \in SI$ we know that they are not on the same process, and they do not send to the same process, so $x \parallel y$ and $v \equiv_{\text{us}} v_0\,y\,x\,v_1$, hence $u' \in \mathcal{W}$.

Now we show that $Cl_{SI}(w|_S) \supseteq \mathcal{W}$ by induction on the size of w. Initially, $\varepsilon|_S = \varepsilon$. Let $w = u\,s\,r$ be a 1-scheduling without $\dashleftarrow$ arc. Consider some $w' \equiv_{\text{us}} w$ such that $w' = x\,s\,y\,r\,z$, with $x, y, z \in Act^*$. We have $w|_S = u|_S\,s$ and $w'|_S = x|_S\,s\,(yz)|_S$. We know that $w \equiv_{\text{us}} xyz\,sr$ so $u \equiv_{\text{us}} xyz$. The case where $w = u\,s$ is dealt with similarly.

By induction, $x|_S(yz)|_S \in Cl_{SI}(u)$, so $x|_S(yz)|_S\,s \in Cl_{SI}(w)$. Now, we know that s is concurrent with every event of yz. Moreover, there is no $\dashleftarrow$ arc. So for every s' occurring in $(yz)|_S$: if $s = p!q(m)$, then s' is not on p, and it is neither a send to p (matched or not) nor to q (matched or not), so $(s', s) \in SI$, and $w'|_S \in Cl_{SI}(w)$. ◀

A viable sequence w is called *good* if there exists a 1-scheduled sequence w' such that $w \equiv_{\text{us}} w'$ and w' has no backward $\dashrightarrow$ arc. Otherwise w is called *bad*. Note that every good viable sequence is necessarily 1-schedulable. A CFM $\mathcal{A}$ is good if every trace in $Tr(\mathcal{A})$ is good, and $\mathcal{A}$ is called bad otherwise.

▶ **Proposition 4.5.** *For every good CFM $\mathcal{A}$, it holds that $Tr(\mathcal{A})|_S$ is the closure under SI of $\{w \in Tr(\mathcal{A}) \mid w \text{ is a 1-scheduling}\}|_S$.*

Proof. Let K and L be defined as $K = K'|_S$ and $L = L'|_S$, where

$$K' = \{w \in Tr(\mathcal{A}) \mid w \text{ is a 1-scheduling and has no backward } \dashrightarrow \text{ arc}\}$$
$$L' = \{w \in Tr(\mathcal{A}) \mid w \text{ is a 1-scheduling}\}$$

Note that $K \subseteq L \subseteq Tr(\mathcal{A})|_S$. We first show that $Tr(\mathcal{A})|_S = Cl_{SI}(K)$. Consider a trace $w \in Tr(\mathcal{A})$. As w is good, there exists a trace $w' \in Tr(\mathcal{A})$ such that $w \equiv_{\text{us}} w'$, w' is a 1-scheduling and w' has no backward $\dashrightarrow$ arc. It holds that $Cl_{SI}(w'|_S) = \{u|_S \mid u \equiv_{\text{us}} w'\}$ according to Lemma 4.4. As $w'|_S \in K$ and $w \equiv_{\text{us}} w'$, we get that $w|_S \in Cl_{SI}(K)$. The converse inclusion $Cl_{SI}(K) \subseteq Tr(\mathcal{A})|_S$ immediately follows from Lemma 4.4. Since $K \subseteq L \subseteq Tr(\mathcal{A})|_S$ and $Tr(\mathcal{A})|_S = Cl_{SI}(K)$, we derive that $Cl_{SI}(K) \subseteq Cl_{SI}(L) \subseteq Cl_{SI}(Cl_{SI}(K)) = Cl_{SI}(K)$, hence, $Tr(\mathcal{A})|_S = Cl_{SI}(K) = Cl_{SI}(L)$. ◀

The above proposition provides our first ingredient to decide the send-synchronizability problem for 1-schedulable CFMs. Two additional ingredients are needed to complete our decidability proof. First, we will show that every bad CFM is not 1-schedulable or not send-synchronizable. Second, badness of a CFM will be shown to be decidable. We start with a characterization of bad viable sequences. For a viable sequence w, a $(\prec \cup \text{msg}^{-1} \cup \dashrightarrow \cup \ll_{\text{us}}^{w})$-cycle in $\texttt{msc}(w)$ is called *trivial* if it is of the form $(e, f, \ldots, e, f)$ with $(e, f) \in (\text{msg} \cup \text{msg}^{-1})$, or equivalently, if it does not contain two distinct sends.

▶ **Lemma 4.6.** *For every viable sequence w, it holds that w is bad if, and only if, there exists a $(\prec \cup \text{msg}^{-1} \cup \dashrightarrow \cup \ll_{\text{us}}^{w})$-cycle in $\texttt{msc}(w)$ that is not trivial.*

Proof sketch. Consider a viable sequence w. For short, let us write *TrvCyc* the condition that every $(\prec \cup \text{msg}^{-1} \cup \dashrightarrow \cup \ll_{\text{us}}^{w})$-cycle in $\texttt{msc}(w)$ is trivial. To prove the lemma, we show that w is good if, and only if, *TrvCyc* holds. We consider the directed graph G whose nodes are the events of $\texttt{msc}(w)$ and whose edges are given by the relations $<_{\mathbb{P}}$, msg, $<_{\texttt{mb}}$, msg^{-1}, $\dashrightarrow$ and $\ll_{\text{us}}^{w}$. Let H denote the directed graph obtained from G by, firstly, merging events that are connected by an msg-arc, and secondly, removing all msg-arcs. It is readily seen that *TrvCyc* is equivalent to the condition that H is acyclic. If *TrvCyc* holds, then the partial order induced by H admits a linearization $x_1 \cdots x_n$. By "unmerging" events in $x_1 \cdots x_n$, we get a linearization w' of $\texttt{msc}(w)$ such that w' is 1-scheduled and w' has no backward $\dashrightarrow$-arc nor backward $\ll_{\text{us}}^{w}$-arc, hence, w is good. Conversely, if w is good then the sequence $x_1 \cdots x_n$ of nodes of H that is induced by w contains every node of H and has no backward edge, so H is acyclic, hence, *TrvCyc* holds. ◀

We now focus our attention to CFMs that are 1-schedulable. For this class of CFMs, we obtain a characterization of badness of a CFM in terms of cycles that are simpler than those of Lemma 4.6 in the sense that they contain no msg^{-1}-arc.

▶ **Lemma 4.7.** *For every 1-schedulable CFM $\mathcal{A}$, if $\mathcal{A}$ is bad then there exists a (bad) trace $w \in Tr(\mathcal{A})$ such that $\texttt{msc}(w)$ contains a $(\prec \cup \dashrightarrow \cup \ll_{\text{us}}^{w})$-cycle.*

Proof. Consider a bad trace $w \in Tr(\mathcal{A})$ of minimal length. Let Ξ denote the set of non-trivial $(<_{\mathbb{P}} \cup \mathrm{msg} \cup <_{\mathtt{mb}} \cup \mathrm{msg}^{-1} \cup \dashrightarrow \cup \ll_{\mathrm{us}}^{w})$-cycles in $\mathtt{msc}(w)$. We lexicographically order cycles ξ in Ξ first by number of msg^{-1}-arcs (accounting for the arc from the last event of ξ to the first event of ξ), second by length of the cycle. By Lemma 4.6, the set Ξ is non-empty. Pick a cycle ξ in Ξ of minimal length, and let us show that ξ contains no msg^{-1}-arc.

We first observe that every event of $\mathtt{msc}(w)$ is in $Past(C)$, where C denotes the set of events of ξ. By contradiction, assume that this is not the case. There necessarily exists an event e in $\mathtt{msc}(w)$ such that $e \notin Past(C)$, e is maximal for $\leq_{\mathbb{P}}$ and e is a receive or an unmatched send. Consider the sequence w' corresponding to w with e removed. It is readily seen that w' is viable, hence, $w' \in Tr(\mathcal{A})$. Moreover, the cycle ξ is still a non-trivial $(\prec \cup \mathrm{msg}^{-1} \cup \dashrightarrow \cup \ll_{\mathrm{us}}^{w})$-cycle in $\mathtt{msc}(w')$. So by Lemma 4.6, w' is a bad trace in $Tr(\mathcal{A})$, which contradicts the minimality of w.

By contradiction, assume that ξ contains an msg^{-1}-arc. Up to a rotation of the cycle, we may assume that ξ is of the form $(e_1, \ldots, e_k)$ with $(e_k, e_1) \in \mathrm{msg}^{-1}$. Note that $k > 2$ since ξ is non-trivial. For readability, let us write $s := e_1$ and $r := e_k$. We make three observations.

1. The event r is maximal for $\leq_{\mathbb{P}}$. By contradiction, suppose that $r <_{\mathbb{P}} f$ for some event f. Since $f \in Past(C)$, we get that $f \preceq e_i$ for some $1 \leq i \leq k$. Pick a $(<_{\mathbb{P}} \cup \mathrm{msg} \cup <_{\mathtt{mb}})$-path $(f_1, \ldots, f_\ell)$ with $f_1 = f$ and $f_\ell = e_i$. We get that $(e_i, \ldots, e_k, f_1, \ldots, f_{\ell-1})$ is in Ξ and has one less msg^{-1}-arc than ξ, which contradicts the minimality of ξ.
2. The event $s' := e_{k-1}$ is a send and $s' <_{\mathbb{P}} r = e_k$. Indeed, as e_k is a receive, e_{k-1} verifies $e_{k-1} <_{\mathbb{P}} r$ or $(e_{k-1}, r) \in \mathrm{msg}$. If $(e_{k-1}, r) \in \mathrm{msg}$ then $e_{k-1} = s = e_1$, hence, $(e_1, \ldots, e_{k-2})$ is in Ξ, which contradicts the minimality of ξ. So $e_{k-1} <_{\mathbb{P}} r$. It remains to show that e_{k-1} is a send. Suppose by contradiction that e_{k-1} is a receive. We either have $e_{k-2} <_{\mathbb{P}} e_{k-1}$ or $(e_{k-2}, e_{k-1}) \in \mathrm{msg}$. In the first case, $e_{k-2} <_{\mathbb{P}} r = e_k$, hence, $(e_1, \ldots, e_{k-2}, e_k)$ is in Ξ, which is impossible by minimality of ξ. In the second case, $(e_{k-2}, e_{k-1}) \in \mathrm{msg}$, hence, $e_{k-2} <_{\mathtt{mb}} s = e_1$. This entails that $(e_1, \ldots, e_{k-2})$ is in Ξ, which is impossible by minimality of ξ.
3. It holds that $s \parallel s'$. Let us prove this claim. We have $s' <_{\mathbb{P}} r$ and $(r, s) \in \mathrm{msg}^{-1}$. Note that $s \neq s'$ since s and s' are sends by distinct processes. If $s \prec s'$ then (s, s', r, s) would be a $(\prec \cup \mathrm{msg}^{-1})$-cycle with at least two distinct sends in $\mathtt{msc}(w)$, which is impossible since w is 1-schedulable. If $s' \prec s$ then $e_{k-1} = s' \prec s = e_1$, hence, $(e_1, \ldots, e_{k-1})$ is in Ξ, which contradicts the minimality of ξ.

We now derive a contradiction, which will conclude the proof that ξ contains no msg^{-1}-arc. Consider the sequence w'' corresponding to w with r removed. We have $w'' \in Tr(\mathcal{A})$ since r is a receive and is maximal for $\leq_{\mathbb{P}}$, by our first observation above. The event $r = e_k$ is the matching receive of $s = e_1$ in $\mathtt{msc}(w)$, so s becomes an unmatched send in $\mathtt{msc}(w'')$. In $\mathtt{msc}(w'')$, the event $s' = e_{k-1}$ is a send and s is a send to the process performing s', by our second observation above. Moreover, as $s \parallel s'$ in $\mathtt{msc}(w)$, by our third observation above, we still have $s \parallel s'$ in $\mathtt{msc}(w'')$. It follows that $s' \dashrightarrow s$ in $\mathtt{msc}(w'')$, hence, $e_{k-1} \dashrightarrow e_1$. We derive that $(e_1, \ldots, e_{k-1})$ is a non-trivial $(\prec \cup \mathrm{msg}^{-1} \cup \dashrightarrow \cup \ll_{\mathrm{us}}^{w})$-cycle in $\mathtt{msc}(w'')$. This entails, by Lemma 4.6, that w'' is bad, which contradicts the minimality of w. ◀

▶ Remark 4.8. Given a viable sequence w, a $(\prec \cup \dashrightarrow \cup \ll_{\mathrm{us}}^{w})$-cycle $(e_1, \ldots, e_k)$ in $\mathtt{msc}(w)$ is called *genuine* if $e_1, \ldots, e_k$ are pairwise-distinct and, for every $1 \leq i < j \leq k$, we have $e_i \prec e_j$ implies $j = i + 1$ and $e_j \prec e_i$ implies $i = 1$ and $j = k$. Informally, a genuine cycle is a simple cycle such that for every events e, f on the cycle, if $e \prec f$ then the subpath of the cycle from e to f is a single $\prec$-arc. It is routinely checked that if $\mathtt{msc}(w)$ contains a $(\prec \cup \dashrightarrow \cup \ll_{\mathrm{us}}^{w})$-cycle then it contains a genuine $(\prec \cup \dashrightarrow \cup \ll_{\mathrm{us}}^{w})$-cycle.

(a) MSC of sequence $w_1 = 3!1(m_0)\,2!3(m_1)\,1!2(m_2)$.

(b) MSC of sequence $w_2 = 4!1(m_0)\,3!4(m_1)\,3!2(m_2)\,2?3(m_2)\,2!1(m_3)$.

Figure 7 Sequences with pattern, and their MSC.

4.2 Bad CFMs

Our goal in this section is to analyze bad, 1-schedulable CFMs. We will show a necessary condition for badness in terms of patterns (Proposition 4.10), and we will show that the presence of a pattern prevents a CFM to be send-synchronizable (Proposition 4.11).

We focus on 1-scheduled traces, as they can be effectively constructed from the CFM using automata. The binary relation $\ll_{\text{us}}^{w}$ needs to be relaxed as we might need to re-order unmatched sends of a 1-scheduling in $Tr(\mathcal{A})$ in order to exhibit a bad 1-schedulable trace in $Tr(\mathcal{A})$. Given a viable sequence w, we introduce the binary relation $\|_{\text{us}}$ over the set of events E of $\mathtt{msc}(w)$, defined by $e \,\|_{\text{us}}\, f$ if there exists $p \in \mathbb{P}$ such that e and f are unmatched sends to p and $e \parallel f$. Note that $\|_{\text{us}}$ is symmetric and contains $\ll_{\text{us}}^{w}$.

A *pattern* for a viable sequence w is an alternating sequence of send events $(x_1,y_1,\ldots,x_k,y_k)$ of $\mathtt{msc}(w)$ satisfying, with the convention that $x_{k+1} := x_1$, the following conditions:

1. $x_i \preceq y_i$, for every $1 \le i \le k$,
2. $y_i \dashrightarrow x_{i+1}$ or $y_i \,\|_{\text{us}}\, x_{i+1}$, for every $1 \le i \le k$,
3. $y_i \dashrightarrow x_{i+1}$ for some $1 \le i \le k$,
4. $x_i \parallel x_j$ and $y_i \parallel y_j$ and $x_i \parallel y_j$, for every $1 \le i, j \le k$ with $i \neq j$.

▶ **Example 4.9.** Figure 7 shows two examples of patterns. In Figure 7a the sequence w_1 has the pattern $(s_0, s_0, s_1, s_1, s_2, s_2)$, with $s_0 \dashrightarrow s_1$, $s_1 \dashrightarrow s_2$ and $s_2 \dashrightarrow s_0$.
The pattern in Figure 7b is (s_0, s_0, s_1, s_3), with $s_0 \dashrightarrow s_1$, $s_1 \prec s_3$, and $s_3 \,\|_{\text{us}}\, s_0$.

▶ **Proposition 4.10.** *Let $\mathcal{A}$ be 1-schedulable CFM. If $\mathcal{A}$ is bad then there exists a trace $w \in Tr(\mathcal{A})$ that admits a pattern $(x_1, y_1, \ldots, x_k, y_k)$.*

Proof. By Lemma 4.7 and Remark 4.8, there exists a trace $w \in Tr(\mathcal{A})$ such that $\mathtt{msc}(w)$ contains a genuine $(\prec \cup \dashrightarrow \cup \ll_{\text{us}}^{w})$-cycle ξ. Observe that ξ necessarily contains a $\dashrightarrow$ arc. The reason is that $\prec$ and $\ll_{\text{us}}^{w}$ are both contained in the total order over positions of w, hence, there is no $(\prec \cup \ll_{\text{us}}^{w})$-cycle in $\mathtt{msc}(w)$. In order to transform $\xi = (e_1, \ldots, e_k)$ into a pattern, we insert "identity" arcs of the form (e_i, e_i) as needed to get an alternating sequence $(x_1, y_1, \ldots, x_k, y_k)$ satisfying the first three conditions of the definition of patterns (recall that $\ll_{\text{us}}^{w}$ is contained in $\|_{\text{us}}$). The fourth (and last) condition holds because ξ is genuine. ◀

▶ **Proposition 4.11.** *Let $\mathcal{A}$ be 1-schedulable. If some 1-scheduling $w \in Tr(\mathcal{A})$ contains a pattern, then $\mathcal{A}$ is not send-synchronizable.*

For the proof of Proposition 4.11 we first provide two technical lemmas. Recall that patterns are cycles. Proposition 4.11 will show a contradiction to send-synchronizability by listing the cycle in two ways. Lemma 4.12 identifies traces corresponding to different listings of the cycle in a such way that the part before the cycle is the same. Lemma 4.13 relates arcs of two equivalent traces with the same send-projection, one of which is rendez-vous.

▶ **Lemma 4.12.** *Let $w \in Tr(\mathcal{A})$ contain some pattern $(x_1, y_1, \ldots, x_k, y_k)$. Let also $Z = \{x_1, y_1, \ldots, x_k, y_k\}$. Then for every $1 \leq i \leq k$ there exist $u, v, u' \in Act^*$ with $uvu' \equiv w$ satisfying the following:*

- *u is a linearization of $Past(Z) \setminus Ftr(Z)$*
- *uv is a linearization of $Past(Z)$*
- *v contains $x_i, y_i, \ldots, x_k, y_k, x_1, y_1, \ldots x_{i-1}, y_{i-1}$ as subsequence.*

Proof. As in Lemma 4.7 we start with a prefix containing all events that are ordered causally before the events of the pattern: u is a linearization of $Past(Z) \setminus Ftr(Z)$.

Consider some $j, \ell \in \{1, \ldots, k\}$, $j \neq \ell$. For readability we write Z_i for the infix $Past(y_i) \cap Ftr(x_i)$ of $\mathtt{msc}(w)$, for every i.

Let also $e \in Z_j$ and $f \in Z_\ell$. If $e \preceq f$ would hold true then, using $f \preceq y_\ell$ and $x_j \preceq e$, we would also get $x_j \preceq y_\ell$, contradicting the definition of pattern. By symmetry, we conclude that $e \parallel f$ for all such e, f.

Therefore, for $1 \leq i \leq k$ we can define v as linearization of $Z_i, Z_{i+1}, \ldots Z_k, Z_1, \ldots Z_{i-1}$. By definition, v contains the subsequence $x_i, y_i, \ldots, x_k, y_k, x_1, y_1, \ldots x_{i-1}, y_{i-1}$. ◀

▶ **Lemma 4.13.** *Consider $w \in Tr(\mathcal{A})$ and $w' \in Tr_{\mathtt{rdv}}(\mathcal{A})$ with $w|_S = w'|_S$. Then for every send events $x \neq y$ in w, with x before y in w:*

- *If $x \dashrightarrow y$ in w, then $x <_{\mathbb{P}} z$, where z is the matching receive of y in w': $(y, z) \in \mathrm{msg}$.*
- *If $x \parallel_{\mathrm{us}} y$ in w, then $x <_{\mathtt{mb}} y$ in w'.*
- *If $x \prec y$ in w then $x \prec y$ in w' by the same causal path.*

Proof. If $x \dashrightarrow y$ in w, we know that x is a send on some process p and y is an unmatched send to process p. Because w' is a rendez-vous execution, with x before y, we get $x <_{\mathbb{P}} z$.

Now suppose that $x \parallel_{\mathrm{us}} y$ in w. Then x and y are two unmatched sends to the same process. The two sends are matched in w', with $x' <_{\mathbb{P}} y'$, where $(x, x') \in \mathrm{msg}, (y, y') \in \mathrm{msg}$. Thus $x \prec y$ in w' because of $<_{\mathtt{mb}}$.

Finally, if $x \prec y$ in w, then $x \prec y$ holds also in w' because every arc of type $<_{\mathbb{P}}, \mathrm{msg}, <_{\mathtt{mb}}$ on the causal path from x to y is preserved in w'. ◀

Proof of Proposition 4.11. We assume that $\mathcal{A}$ is send-synchronizable, and consider some $w \in Tr(\mathcal{A})$ with pattern $(x_1, y_1, \ldots, x_k, y_k)$. We will assume that the pattern is minimal w.r.t. the following parameter: the size of the pattern is the sum of the lengths of the shortest paths from x_j to y_j, over all j.

By Lemma 4.12 we can assume that $w = uv$ is a linearization of $Past(Z)$, with $Z = \{x_1, y_1, \ldots, x_k, y_k\}$, and $x_1, y_1, \ldots, x_k, y_k$ occurs as subsequence of v. In addition, u is linearization of $Past(Z) \setminus Ftr(Z)$.

We know that $(x_1, y_1, \ldots, x_k, y_k)$ is a pattern in v, and we can assume w.l.o.g. that $y_k \dashrightarrow x_1$. Note that $k > 1$ since $x_1 \preceq y_1$ (by definition of patterns) and $x_1 \parallel y_k$ (by definition of $\dashrightarrow$). By Lemma 4.12, for every $1 \leq i \leq k$ there exists some $v' \equiv v$ such that $x_{i+1}, y_{i+1}, \ldots, x_k, y_k, x_1, y_1, \ldots, x_i, y_i$ is a subsequence of v'. Chose for example $i = 1$.

As $\mathcal{A}$ is send-synchronizable, there exist $z, z' \in Tr_{\mathtt{rdv}}(\mathcal{A})$ such that $(uv)|_S = z|_S$ and $(uv')|_S = z'|_S$. From Lemma 4.13, we get that there exists a $(\prec \cup \mathrm{msg}^{-1})^*$-path from x_1 to y_k in z not going through $rcv(x_1)$, the receive matching x_1. Indeed, for each $1 \leq i \leq k$, we have $x_i \prec y_i$ in w, so we still have it in z and it does not go through $rcv(x_1)$. We also have $y_i \dashrightarrow x_{i+1}$ or $y_i \parallel_{\mathrm{us}} x_{i+1}$ in w, for every i. If $y_i \dashrightarrow x_{i+1}$ then $y_i <_{\mathbb{P}} rcv(x_{i+1})\, \mathrm{msg}^{-1}\, x_{i+1}$ in z. If $y_i \parallel_{\mathrm{us}} x_{i+1}$, then $y_i <_{\mathtt{mb}} x_{i+1}$ in z. In both cases the arcs do not use $rcv(x_1)$.

We inspect now $M := \mathtt{msc}(z)$. Let p be the process of the send y_k (recall that it is also the process of $rcv(x_1)$, because $y_k \dashrightarrow x_1$). Because of pattern minimality, we can show that the only send event *on* p in v is y_k. Suppose that some $s \neq y_k$ is a send on process p with

s in v. By definition of u, v, we have $x_k \preceq s \prec y_k$. But then the pattern $(x_1, y_1, \ldots, x_k, s)$ would be smaller, contradiction. Again by minimality, we can show that the only send *to* p in v is x_1. First there is no matched send to p in v, because such a send s would satisfy $s \prec x_1$ as x_1 is an unmatched send to p, hence, if $x_j \preceq s \preceq y_j$ with $1 \leq j \leq k$, then $x_j \prec x_1$, which is forbidden by the definition of patterns. Now if we had some $s \neq x_1$ unmatched send event to p, with say $x_j \preceq s \preceq y_j$ and $1 \leq j < k$, then the pattern $(s, y_j, \ldots, x_k, y_k)$ would be smaller. Finally, if $j = k$, so $x_k \preceq s \prec y_k$, then the predecessor of s on the causal path from x_k to s is an event on p, and we can shorten path from x_k to y_k, obtaining a smaller pattern.

Let us construct $z^\sharp$ from z by moving $rcv(x_1)$ at the end of the sequence. Observe that $z = match(z|_S) = match(u|_S)match(v|_S)$ and similarly $z' = match(u|_S)match(v'|_S)$, where $match : S^* \to Act^*$ is the morphism defined by $match(p!q(m)) = p!q(m)\, q?p(m)$. We derive from the previous paragraph that the sequence of actions by p is $rcv(x_1)y_k$ in $match(v|_S)$ and $y_k rcv(x_1)$ in $match(v'|_S)$. As $rcv(x_1)$ is the last receive by p in z, moving it at the end of the sequence preserves viability, so $z^\sharp$ is viable. Moreover, the sequence of actions by p in $z^\sharp$ coincides with that of z', and the sequence of actions by $q \neq p$ in $z^\sharp$ coincides with that of z. Since $z \in Tr(\mathcal{A})$ and $z' \in Tr(\mathcal{A})$, we get that $z^\sharp \in Tr(\mathcal{A})$.

However, note that now there is a $(\prec \cup \mathrm{msg}^{-1})$-cycle with at least two distinct sends in $M^\sharp := \mathtt{msc}(z^\sharp)$. Indeed, we still have in $M^\sharp$ the $(\prec \cup \mathrm{msg}^{-1})^+$ -path of M from x_1 to y_k, as this path does not go through $rcv(x_1)$, and $y_k <_{\mathbb{P}} rcv(x_1)$ in $M^\sharp$, finishing the cycle with $(rcv(x_1), x_1) \in \mathrm{msg}^{-1}$. So $z^\sharp$ is not 1-schedulable because it contains a non-trivial $(\prec \cup \mathrm{msg}^{-1})^+$-cycle with at least two sends, which contradicts our assumption that $\mathcal{A}$ is 1-schedulable. ◀

4.3 Wrap-Up

In this section, we prove that send-synchronizability is decidable for 1-schedulable CFMs, by assembling the results from the previous subsections. We start with the following observation, which is an immediate consequence of Propositions 4.5, 4.10, and 4.11.

▶ **Proposition 4.14.** *For every 1-schedulable CFM $\mathcal{A}$, it holds that $\mathcal{A}$ is send-synchronizable if, and only if, (1) $\mathcal{A}$ is good, (2) $Tr_{rdv}(\mathcal{A})|_S = \{w \in Tr(\mathcal{A}) \mid w \text{ is a 1-scheduling}\}|_S$ and (3) $Tr_{rdv}(\mathcal{A})|_S$ is closed under SI.*

▶ **Proposition 4.15.** *It is decidable to check if a 1-schedulable CFM is good.*

Proof sketch. Let $\mathcal{A}$ be a 1-schedulable CFM. From Proposition 4.10, we know that $\mathcal{A}$ is bad if there exists a trace with a pattern. The existence of a pattern can be checked on a 1-scheduling, as by construction two equivalent traces have the same patterns, and $\mathcal{A}$ is 1-schedulable. Moreover the set of 1-scheduling of $\mathcal{A}$ is regular.

We guess a bad 1-scheduling on-the-fly, together with the sends that form the pattern. We can assume that the pattern occurs in the order of the sequence. If there exists some bad trace in $Tr(\mathcal{A})$, then there is one with a pattern of size at most $2 \times |\mathbb{P}|$, as otherwise there would be at least three sends of the pattern on the same process, and we would have a shortcut.

Let $(x_1, y_1, \ldots, x_k, y_k)$ be the pattern we read. We need to check that

- for each $0 \leq i \leq k$, $x_i \preceq y_i$ and $y_i \parallel x_{i+1}$ with either $\dashrightarrow$ or $\parallel_{\mathrm{us}}$, and at least one $\dashrightarrow$-arc.
- and for each $0 \leq i, j \leq k$ with $i \neq j$, $x_i \parallel x_j$ and $y_i \parallel y_j$ and $x_i \parallel y_j$.

To do so, we will keep for each send s of the pattern the set of processes with actions causally ordered after s, and we will update these sets while reading the bad 1-scheduling. So we need a memory of size $O(|\mathbb{P}|^2)$. ◀

From the two previous propositions, we get the following theorem.

▶ **Theorem 4.16.** *The question whether a 1-schedulable CFM is send-synchronizable is* PSPACE*-complete.*

Proof. Let $\mathcal{A}$ be a 1-schedulable CFM. From Proposition 4.15, we know that we can check in polynomial space if $\mathcal{A}$ is good. The two sets $Tr_{\texttt{rdv}}(\mathcal{A})$ and $\{w \in Tr(\mathcal{A}) \mid w \text{ is a 1-scheduling}\}$ are regular, so we can check in PSPACE if they are equal. And from Remark 4.3, we know that we can check if $Tr_{\texttt{rdv}}(\mathcal{A})|_S$ is closed under SI in PSPACE.

For PSPACE-hardness, we reduce from the automata intersection problem as in [8], with a send that cannot be received if the intersection of the automata is not empty. ◀

5 Conclusion

We showed that send-synchronizability for mailbox CFMs is undecidable, thus closing the question left open in [11, 9]. We also showed that we can decide in PSPACE if a 1-schedulable CFM (which is a property that can be checked in PSPACE) is send-synchronizable. We think that the techniques developed for the proof could be used to answer others problems for mailbox CFMs, like the realizability problem. Moreover, even if the class of 1-schedulable systems is decidable, it is quite restrictive. It can be interesting to see if our techniques can be used for k-schedulable CFMs and the more general class of CFMs studied in [8].

References

1 Parosh Aziz Abdulla and Bengt Jonsson. Verifying programs with unreliable channels. *Information and Computation*, 127(2):91–101, 1996. doi:10.1006/INCO.1996.0053.

2 Samik Basu and Tevfik Bultan. Choreography conformance via synchronizability. In Sadagopan Srinivasan, Krithi Ramamritham, Arun Kumar, M. P. Ravindra, Elisa Bertino, and Ravi Kumar, editors, *Proceedings of the 20th International Conference on World Wide Web, WWW 2011, Hyderabad, India, March 28 - April 1, 2011*, pages 795–804. ACM, 2011. doi:10.1145/1963405.1963516.

3 Samik Basu and Tevfik Bultan. On deciding synchronizability for asynchronously communicating systems. *Theoretical Computer Science*, 656:60–75, 2016. doi:10.1016/J.TCS.2016.09.023.

4 Samik Basu, Tevfik Bultan, and Meriem Ouederni. Synchronizability for verification of asynchronously communicating systems. In Viktor Kuncak and Andrey Rybalchenko, editors, *Verification, Model Checking, and Abstract Interpretation - 13th International Conference, VMCAI 2012, Philadelphia, PA, USA, January 22-24, 2012. Proceedings*, volume 7148 of *Lecture Notes in Computer Science*, pages 56–71. Springer, 2012. doi:10.1007/978-3-642-27940-9_5.

5 Benedikt Bollig, Cinzia Di Giusto, Alain Finkel, Laetitia Laversa, Étienne Lozes, and Amrita Suresh. A unifying framework for deciding synchronizability. In *32nd International Conference on Concurrency Theory, CONCUR 2021, August 24-27, 2021, Virtual Conference*, volume 203 of *LIPIcs*, pages 14:1–14:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICS.CONCUR.2021.14.

6 Ahmed Bouajjani, Constantin Enea, Kailiang Ji, and Shaz Qadeer. On the completeness of verifying message passing programs under bounded asynchrony. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*, volume 10982 of *Lecture Notes in Computer Science*, pages 372–391. Springer, 2018. doi:10.1007/978-3-319-96142-2_23.

7 Daniel Brand and Pitro Zafiropulo. On communicating finite-state machines. *Journal of the ACM*, 30(2):323–342, 1983. doi:10.1145/322374.322380.

8 Romain Delpy, Anca Muscholl, and Grégoire Sutre. An automata-based approach for synchronizable mailbox communication. In *35th International Conference on Concurrency Theory, CONCUR 2024, September 9-13, 2024, Calgary, Canada*, volume 311 of *LIPIcs*, pages 22:1–22:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPICS.CONCUR.2024.22.

9 Cinzia Di Giusto, Laetitia Laversa, and Kirstin Peters. Synchronisability in mailbox communication. In Proceedings Combined 31st International Workshop on *Theoretical Computer Science Expressiveness in Concurrency*and 21st Workshop on *Structural Operational Semantics,* Calgary, Canada, 9th September 2024, volume 412 of *Electronic Proceedings in Theoretical Computer Science*, pages 19–34. Open Publishing Association, 2024. doi:10.4204/EPTCS.412.3.

10 Alain Finkel and Étienne Lozes. Synchronizability of communicating finite state machines is not decidable. In Ioannis Chatzigiannakis, Piotr Indyk, Fabian Kuhn, and Anca Muscholl, editors, *44th International Colloquium on Automata, Languages, and Programming, ICALP 2017, July 10-14, 2017, Warsaw, Poland*, volume 80 of *LIPIcs*, pages 122:1–122:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPICS.ICALP.2017.122.

11 Alain Finkel and Étienne Lozes. Synchronizability of communicating finite state machines is not decidable. *Logical Methods in Computer Science*, 19(4), 2023. doi:10.46298/LMCS-19(4:33)2023.

12 Alain Finkel and Philippe Schnoebelen. Well-structured transition systems everywhere! *Theoretical Computer Science*, 256(1-2):63–92, 2001. doi:10.1016/S0304-3975(00)00102-X.

13 Xiang Fu, Tevfik Bultan, and Jianwen Su. Synchronizability of conversations among web services. *IEEE Transactions on Software Engineering*, 31(12):1042–1055, 2005. doi:10.1109/TSE.2005.141.

14 Blaise Genest, Dietrich Kuske, and Anca Muscholl. On communicating automata with bounded channels. *Fundamenta Informaticae*, 80(1-3):147–167, 2007. URL: http://content.iospress.com/articles/fundamenta-informaticae/fi80-1-3-09.

15 Cinzia Di Giusto, Davide Ferré, Laetitia Laversa, and Étienne Lozes. A partial order view of message-passing communication models. *Proceedings of the ACM on Programming Languages*, 7(POPL):1601–1627, 2023. doi:10.1145/3571248.

16 Cinzia Di Giusto, Laetitia Laversa, and Étienne Lozes. On the k-synchronizability of systems. In *Foundations of Software Science and Computation Structures - 23rd International Conference, FOSSACS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings*, volume 12077 of *Lecture Notes in Computer Science*, pages 157–176. Springer, 2020. doi:10.1007/978-3-030-45231-5_9.

17 Cinzia Di Giusto, Laetitia Laversa, and Étienne Lozes. Guessing the buffer bound for k-synchronizability. *International Journal of Foundations of Computer Science*, 34(8):1051–1076, 2023. doi:10.1142/S0129054122430018.

18 Jesper G. Henriksen, Madhavan Mukund, K. Narayan Kumar, Milind A. Sohoni, and P. S. Thiagarajan. A theory of regular MSC languages. *Information and Computation*, 202(1):1–38, 2005. doi:10.1016/J.IC.2004.08.004.

19 John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

20 Dietrich Kuske and Anca Muscholl. Communicating automata. In *Handbook of Automata Theory*, pages 1147–1188. European Mathematical Society Publishing House, Zürich, Switzerland, 2021. doi:10.4171/AUTOMATA-2/9.

21 Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978. doi:10.1145/359545.359563.

22 Elaine Li, Felix Stutz, Thomas Wies, and Damien Zufferey. Complete multiparty session type projection with automata. In *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part III*, volume 13966 of *Lecture*

Notes in Computer Science, pages 350–373. Springer, 2023. doi:10.1007/978-3-031-37709-9_17.

23 Rupak Majumdar, Madhavan Mukund, Felix Stutz, and Damien Zufferey. Generalising projection in asynchronous multiparty session types. In *32nd International Conference on Concurrency Theory, CONCUR 2021, August 24-27, 2021, Virtual Conference*, volume 203 of *LIPIcs*, pages 35:1–35:24. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICS.CONCUR.2021.35.

24 Felix Stutz. Asynchronous multiparty session type implementability is decidable - lessons learned from message sequence charts. In *37th European Conference on Object-Oriented Programming, ECOOP 2023, July 17-21, 2023, Seattle, Washington, United States*, volume 263 of *LIPIcs*, pages 32:1–32:31. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICS.ECOOP.2023.32.

A 1-schedulability and related notions

Similar notions to 1-schedulability can be found in several papers [5, 17]. The way unmatched sends are treated in the literature may differ in a significant way. Our notion of 1-schedulable is the same as strongly 1-synchronisable in [5]. The notion of RSC in [15] is more restrictive than 1-schedulable, as it does not allow to permute two unmatched sends to the same process.

Take as an example the trace 0!2(a) 0!1(b) 2!1(c) 2?0(a). It is 1-schedulable since it is equivalent to 2!1(c) 0!2(a) 2?0(a) 0!1(b). But it is not RSC according to [15], since RSC disallows permuting 0!1(b), 2!1(c). Note however that every 1-scheduling is an RSC execution.

B A necessary and sufficient condition for send-synchronizability

Let $\mathcal{A}$ be a CFM. Recall that $\mathcal{A}$ is send-synchronizable if, and only if, for every trace $u \in Tr(\mathcal{A})$, there exists a trace $v \in Tr_{\texttt{rdv}}(\mathcal{A})$ with $u|_S = v|_S$. In fact, the trace v is unique if it exists. Indeed, the only candidate for v is $match(u|_S)$. Recall that $match : S^* \to Act^*$ is the morphism defined by $match(p!q(m)) = p!q(m)\, q?p(m)$. Observe that $match(u)|_S = u$ for all $u \in S^*$. This leads us to the characterization of send-synchronizability in Lemma B.1 below. The following notations are used in this lemma. For every process $p \in \mathbb{P}$, $L(\mathcal{A}_p)$ is the language of the LTS $\mathcal{A}_p$ (viewed as a non-deterministic automaton with every state final) and $u_{|p}$ is the projection of $u \in Act^*$ on Act_p.

▶ **Lemma B.1.** *A CFM $\mathcal{A}$ is send-synchronizable if, and only if, for every trace $u \in Tr(\mathcal{A})$, the sequence $v = match(u|_S)$ satisfies $v_{|p} \in L(\mathcal{A}_p)$ for every process $p \in \mathbb{P}$.*

Proof. Assume that $\mathcal{A}$ is send-synchronizable. Let $u \in Tr(\mathcal{A})$ and define $v = match(u|_S)$. Since $\mathcal{A}$ is send-synchronizable, there exists $u' \in Tr_{\texttt{rdv}}(\mathcal{A})$ such that $u|_S = u'|_S$. We have $match(u'|_S) = u'$ since $u' \in Tr_{\texttt{rdv}}(\mathcal{A})$. It follows that $u' = v$, hence, $v \in Tr_{\texttt{rdv}}(\mathcal{A})$. This entails that $v_{|p} \in L(\mathcal{A}_p)$ for every process $p \in \mathbb{P}$.

Conversely, let $u \in Tr(\mathcal{A})$ and assume that $v = match(u|_S)$ satisfies $v_{|p} \in L(\mathcal{A}_p)$ for every process $p \in \mathbb{P}$. The sequence v is clearly rendez-vous, according to the definition of *match*. We derive that $v \in Tr_{\texttt{rdv}}(\mathcal{A})$, since $v_{|p} \in L(\mathcal{A}_p)$ for every process $p \in \mathbb{P}$. The observation that $v|_S = match(u|_S)|_S = u|_S$ entails that $u|_S \in Tr_{\texttt{rdv}}(\mathcal{A})|_S$, which concludes the proof. ◀

C Proofs of Section 3

▶ **Lemma 3.1.** *If $\mathcal{I}$ is a positive instance of Pre-MPCP then there is an initial execution in $\mathcal{T}(\mathcal{A}(\mathcal{I}))$ leading to the configuration $(N, N, N; \varepsilon, \varepsilon, \varepsilon)$.*

Proof. Consider a sequence of indices $i_1, \ldots, i_k \in \{1, \ldots, K\}$, with $k \geq 1$, such that $i_1 = 1$, $x_{i_1} \cdots x_{i_k} = y_{i_1} \cdots y_{i_k}$ and $|x_{i_1} \cdots x_{i_j}| \geq |y_{i_1} \cdots y_{i_j}|$ for every $j \in \{1, \ldots, k\}$. Let us write $a_1 \cdots a_n$, with $a_i \in \Sigma$, the word $x_{i_1} \cdots x_{i_k}$ (or, equivalently, the word $y_{i_1} \cdots y_{i_k}$). Define $e_j = |x_{i_1} \cdots x_{i_j}|$ and $f_j = |y_{i_1} \cdots y_{i_j}|$, for every $j \in \{1, \ldots, k\}$. We also put $e_0 = f_0 = 0$. Our assumptions on $i_1, \ldots, i_k$ entail, firstly, that $e_k = f_k = n$, and, secondly, that $e_j \geq f_j$, $x_{i_j} = a_{e_{j-1}+1} \cdots a_{e_j}$ and $y_{i_j} = a_{f_{j-1}+1} \cdots a_{f_j}$, for all $j \in \{1, \ldots, k\}$.

We show that for every $j \in \{1, \ldots, k\}$, there is an execution ρ_j in $\mathcal{T}(\mathcal{A}(\mathcal{I}))$ from the configuration $(N, N, N; \varepsilon, v_j^R, v_j^V)$, with $v_j^R = a_{f_{j-1}+1} \cdots a_{e_{j-1}}$ and $v_j^V = a_1\overline{a_1} \cdots a_{f_{j-1}}\overline{a_{f_{j-1}}}$, to the configuration $(N, N, N; \varepsilon, w_j^R, w_j^V)$, with $w_j^R = a_{f_j+1} \cdots a_{e_j}$ and $w_j^V = a_1\overline{a_1} \cdots a_{f_j}\overline{a_{f_j}}$. In the execution ρ_j, the sequence of actions performed by G is $G!R(x_{i_j})\, G!V(\overline{y_{i_j}})$, the sequences of receive and send actions performed by R are $R?G(y_{i_j})$ and $R!V(y_{i_j})$, respectively, and V does not move. These sequences of actions are permitted by the LTSes $\mathcal{A}_G$ and $\mathcal{A}_R$ (see Figures 2 and 3). To make it clear that such an execution ρ_j exists, we need to show that these actions can be interleaved in a way that produces an viable sequence (that is, in addition, permitted by the LTS $\mathcal{A}_R$). To reduce clutter, we define $b_1 = a_{f_{j-1}+1}, \ldots, b_m = a_{f_j}$, where $m = f_j - f_{j-1}$. Note that $y_{i_j} = b_1 \cdots b_m$. Formally, the viable sequence labeling the execution ρ_j is

$$G!R(x_{i_j}) \cdot R?G(b_1)\, R!V(b_1)\, G!V(\overline{b_1}) \cdots R?G(b_m)\, R!V(b_m)\, G!V(\overline{b_m})$$

This sequence is viable because the contents z of R's mailbox after $G!R(x_{i_j})$ verifies $z = v_j^R \cdot x_{i_j} = a_{f_{j-1}+1} \cdots a_{e_j}$, hence, z admits $b_1 \cdots b_m$ as prefix, since $b_1 \cdots b_m = a_{f_{j-1}+1} \cdots a_{f_j}$ and $e_j \geq f_j$. It is readily seen that the configuration $(N, N, N; \varepsilon, w_j^R, w_j^V)$ obtained after ρ_j verifies $w_j^R = a_{f_j+1} \cdots a_{e_j}$ and $w_j^V = a_1\overline{a_1} \cdots a_{f_j}\overline{a_{f_j}}$.

We also observe that there is an execution ρ_{k+1} in $\mathcal{T}(\mathcal{A}(\mathcal{I}))$ from the configuration $(N, N, N; \varepsilon, \varepsilon, v_k^V)$, with $v_k^V = a_1\overline{a_1} \cdots a_n\overline{a_n}$, to the configuration $(N, N, N; \varepsilon, \varepsilon, \varepsilon)$. The trace of ρ_{k+1} is the viable sequence $V?R(a_1)\, V?G(\overline{a_1}) \cdots V?R(a_n)\, V?G(\overline{a_n})$. This sequence of actions is permitted by the LTS $\mathcal{A}_V$ (see Figure 4).

By concatenating the executions $\rho_1, \ldots, \rho_{k+1}$, we get an execution from $(N, N, N; \varepsilon, \varepsilon, \varepsilon)$ to $(N, N, N; \varepsilon, \varepsilon, \varepsilon)$. But this execution does not start from the initial configuration of $\mathcal{T}(\mathcal{A}(\mathcal{I}))$. Along the execution ρ_1, G does $G!R(x_1)G!V(\overline{y_1})$, since $i_1 = 1$. So G could as well have started from its initial state. Obviously, the same goes for R and V, since they can move silently from their initial state to the state N. Hence, there exists an execution ρ_1' from the initial configuration of $\mathcal{T}(\mathcal{A}(\mathcal{I}))$ to the configuration obtained after ρ_1. The concatenation $\rho_1'\rho_2 \cdots \rho_K$ provides an initial execution leading to the configuration $(N, N, N; \varepsilon, \varepsilon, \varepsilon)$. ◀

▶ **Corollary 3.2.** *If $\mathcal{I}$ is a positive instance of* Pre-MPCP *then there is a trace $u \in Tr(\mathcal{A}(\mathcal{I}))$ containing the action $R!V(\#)$, hence, $\mathcal{A}(\mathcal{I})$ is not send-synchronizable.*

Proof. Starting from the configuration $(N, N, N; \varepsilon, \varepsilon, \varepsilon)$, there is an execution in $\mathcal{T}(\mathcal{A}(\mathcal{I}))$ labelled by

$$G!R(\$)\, R?G(\$)\, R!V(\$)\, G!V(\overline{\$})\, V?R(\$)\, V?G(\overline{\$})\, V!G(\#)\, G?V(\#)\, G!R(\#)\, R?G(\#)\, R!V(\#).$$

If $\mathcal{I}$ is a positive instance of Pre-MPCP then, according to Lemma 3.1 and the above observation, there is a trace $u \in Tr(\mathcal{A}(\mathcal{I}))$ ending with $R!V(\#)$. Obviously, it holds that $u|_S \in Tr(\mathcal{A}(\mathcal{I}))|_S$. But, as V contains no $V?R(\#)$ transition, no trace in $Tr_{\texttt{rdv}}(\mathcal{A}(\mathcal{I}))$ contains the action $R!V(\#)$. Hence, $u|_S \notin Tr_{\texttt{rdv}}(\mathcal{A}(\mathcal{I}))|_S$, which entails that $\mathcal{A}(\mathcal{I})$ is not send-synchronizable. ◀

▶ **Lemma 3.3.** *If there is a trace in $Tr(\mathcal{A}(\mathcal{I}))$ containing the action $R!V(\#)$ then $\mathcal{I}$ is a positive instance of* Pre-MPCP.

Proof. Consider an initial execution ρ in $\mathcal{T}(\mathcal{A}(\mathcal{I}))$ ending with the action $R!V(\#)$. By construction, ρ also necessarily contains $R?G(\#)$, hence, it also contains $G!R(\#)$ and $G?V(\#)$, which entails that it contains $V!G(\#)$. Therefore, in the execution ρ, the processes R and V have each selected the normal mode, since there is no $R!V(\#)$ transition nor $V!G(\#)$ transition in the dummy modes. Let π denote the prefix of ρ that ends just before the $V!G(\#)$ action, and let u denote the trace labelling π. According to the definition of the LTSes $\mathcal{A}_G$, $\mathcal{A}_R$ and $\mathcal{A}_V$ (see Figures 2–4), the projections $u_{|G}$, $u_{|R}$ and $u_{|V}$ are as follows:

$$\begin{aligned} u_{|G} &= G!R(x_{i_1})\,G!V(\overline{y_{i_1}}) \cdots G!R(x_{i_k})\,G!V(\overline{y_{i_k}}) \cdot G!R(\$)\,G!V(\overline{\$}) \\ u_{|R} &= R?G(b_1)\,R!V(b_1) \cdots R?G(b_m)\,R!V(b_m) \cdot \alpha \\ u_{|V} &= V?R(a_1)\,V?G(\overline{a_1}) \cdots V?R(a_n)\,V?G(\overline{a_n}) \cdot V?R(\$)\,V?G(\overline{\$}) \end{aligned}$$

for some $i_1, \ldots, i_k \in \{1, \ldots, K\}$ with $k \geq 1$ and $i_1 = 1$, some $a_1, \ldots, a_n \in \Sigma$ with $n \geq 0$, some $b_1, \ldots, b_m \in (\Sigma \cup \{\$, \#\})$ with $m \geq 0$, and some α that is either ε or a receive action $R?G(\cdot)$. The word $a_1\overline{a_1} \cdots a_n\overline{a_n}\$\overline{\$}$ received by V is a prefix of a shuffle of the word $b_1 \cdots b_m$ sent by R to V and of the word $\overline{y_{i_1}} \cdots \overline{y_{i_k}}\overline{\$}$ sent by G to V. So $a_1 \cdots a_n\$$ is a prefix of $b_1 \cdots b_m$ and $\overline{a_1} \cdots \overline{a_n}\overline{\$}$ is a prefix of $\overline{y_{i_1}} \cdots \overline{y_{i_k}}\overline{\$}$. Since $\overline{\$}$ does not occur in $\overline{y_{i_1}} \cdots \overline{y_{i_k}}$, we get that $a_1 \cdots a_n = y_{i_1} \cdots y_{i_k}$. Similarly, the word $b_1 \cdots b_m$ received by R is a prefix of the word $x_{i_1} \cdots x_{i_k}\$$ sent by G to R. We derive that $a_1 \cdots a_n\$$ is a prefix of $x_{i_1} \cdots x_{i_k}\$$. Since $\$$ does not occur in $x_{i_1} \cdots x_{i_k}$, this entails that $a_1 \cdots a_n\$ = b_1 \cdots b_m = x_{i_1} \cdots x_{i_k}\$$. We have shown that $x_{i_1} \cdots x_{i_k} = y_{i_1} \cdots y_{i_k}$. It remains to prove that $|x_{i_1} \cdots x_{i_j}| \geq |y_{i_1} \cdots y_{i_j}|$ for all $j \in \{1, \ldots, k\}$. Before that, we observe that every send action to V is matched in u. Indeed, the length of the word $a_1\overline{a_1} \cdots a_n\overline{a_n}\$\overline{\$}$ received by V is $2(n+1)$, and the length of the word sent to V is $|b_1 \cdots b_m| + |\overline{y_{i_1}} \cdots \overline{y_{i_k}}\overline{\$}| = 2m = 2(n+1)$. This means that the sequence of send actions to V in u is $R!V(b_1)\,G!V(\overline{b_1}) \cdots R!V(b_m)\,G!V(\overline{b_m})$.

We now show that $|x_{i_1} \cdots x_{i_j}| \geq |y_{i_1} \cdots y_{i_j}|$ for all $j \in \{1, \ldots, k\}$. Let $j \in \{1, \ldots, k\}$ and consider a prefix u_j of u such that $u_{j|G} = G!R(x_{i_1})\,G!V(\overline{y_{i_1}}) \cdots G!R(x_{i_j})\,G!V(\overline{y_{i_j}})$. The projection of u_j on Act_R may be written as $u_{j|R} = R?G(b_1)\,R!V(b_1) \cdots R?G(b_h)\,R!V(b_h) \cdot \beta$ for some $h \in \{0, \ldots, m\}$ and some β that is either ε or a receive action $R?G(\cdot)$. The word $b_1 \cdots b_h$ received by R is a prefix of the word $x_{i_1} \cdots x_{i_j}$ sent by G to R, so $|x_{i_1} \cdots x_{i_j}| \geq h$. Now, the word sent to V in u_j, say z, is a shuffle of the word $b_1 \cdots b_h$ sent by R to V and of the word $\overline{y_{i_1}} \cdots \overline{y_{i_j}}$ sent by G to V. The word z is also a prefix of the word $b_1\overline{b_1} \cdots b_m\overline{b_m}$ sent to V in u. This entails that $|b_1 \cdots b_h| = |\overline{y_{i_1}} \cdots \overline{y_{i_j}}|$ or $|b_1 \cdots b_h| = |\overline{y_{i_1}} \cdots \overline{y_{i_j}}| + 1$. It follows that $h \geq |y_{i_1} \cdots y_{i_j}|$ and we conclude that $|x_{i_1} \cdots x_{i_j}| \geq h \geq |y_{i_1} \cdots y_{i_j}|$. ◀

▶ **Lemma 3.4.** *For every trace $u \in Tr(\mathcal{A}(\mathcal{I}))$, if u does not contain the action $R!V(\#)$ then $match(u|_S)_{|p} \in L(\mathcal{A}_p)$ for every process $p \in \{G, R, V\}$.*

Proof. Let $u \in Tr(\mathcal{A}(\mathcal{I}))$ such that u does not contain the action $R!V(\#)$. We consider each process in $\mathbb{P} = \{G, R, V\}$ separately. Recall that the LTSes $\mathcal{A}_G$, $\mathcal{A}_R$ and $\mathcal{A}_V$ of the processes G, R and V are defined in Figures 2–4.

Let us start with the guesser process. According to the definition of $\mathcal{A}_G$, there exists a sequence of indices $i_1, \ldots, i_k \in \{1, \ldots, K\}$, with $k \geq 1$ and $i_1 = 1$, such that the sequence of send actions performed by G in u is a prefix of $v \cdot G!R(\$)\,G!V(\overline{\$})\,G!R(\#)$, where $v = G!R(x_{i_1})\,G!V(\overline{y_{i_1}}) \cdots G!R(x_{i_k})\,G!V(\overline{y_{i_k}})$. According to the definition of $\mathcal{A}_R$ and $\mathcal{A}_V$, the sequence of send actions to G in u is either empty or the single action $V!G(\#)$. Moreover, if

u contains $V!G(\#)$ then this action is necessarily after $G!V(\overline{\$})$ and before $G!R(\#)$. This is due to the receive actions $V?G(\overline{\$})$ and $G?V(\#)$. We derive that $match(u|_S)_{|G}$ is a prefix of $v \cdot G!R(\$)\, G!V(\overline{\$})\, G?V(\#)\, G!R(\#)$ and conclude that $match(u|_S)_{|G} \in L(\mathcal{A}_G)$.

We now proceed to the relayer process. According to the definition of $\mathcal{A}_G$ and $\mathcal{A}_V$, the sequence of send actions to R in u is a prefix of a sequence of the form $G!R(a_1) \cdots G!R(a_n) \cdot G!R(\$)\, G!R(\#)$, with $n \geq 0$ and $a_i \in \Sigma$. So the word sent to R in u is a prefix of $a_1 \cdots a_n\$\#$. This entails that the sequence of send actions performed by R in u is in the language $\{R!V(a) \mid a \in \Sigma\}^* \cdot \{\varepsilon, R!V(\$)\}$, according to the definition of $\mathcal{A}_R$. Indeed, either R selects the normal mode, in which case the word sent by R to V is a prefix of the word $a_1 \cdots a_n\$$ (recall that u does not contain the action $R!V(\#)$ by assumption), or R selects the dummy mode, in which case the word sent by R to V is in the language $\Sigma^* \cdot \{\varepsilon, \$\}$. We derive that the projection of u on sends by R, or to R, is in the shuffle of the language $\{R!V(a) \mid a \in \Sigma\}^* \cdot \{\varepsilon, R!V(\$)\}$ and the language $\{G!R(m) \mid m \in \Sigma \cup \{\$, \#\}\}^*$. It follows that $match(u|_S)_{|R}$ is in the shuffle of the language $\{R!V(a) \mid a \in \Sigma\}^* \cdot \{\varepsilon, R!V(\$)\}$ and the language $\{R?G(m) \mid m \in \Sigma \cup \{\$, \#\}\}^*$. We conclude that $match(u|_S)_{|R} \in L(\mathcal{A}_R)$, by letting R select the dummy mode.

Let us finally address the verifier process. We consider two cases, depending on whether u contains the action $V!G(\#)$ or not. If u does not contain the action $V!G(\#)$, then V does not perform any send action in u, according to the definition of $\mathcal{A}_V$. The projection of u on sends to V is in the language $(\{G!V(\overline{m}) \mid m \in \Sigma \cup \{\$\}\} \cup \{R!V(m) \mid m \in \Sigma \cup \{\$\}\})^*$, according to the definition of $\mathcal{A}_G$ and $\mathcal{A}_R$ (recall that u does not contain the action $R!V(\#)$ by assumption). It follows that $match(u|_S)_{|V}$ is in the language $(\{V?G(\overline{m}) \mid m \in \Sigma \cup \{\$\}\} \cup \{V?R(m) \mid m \in \Sigma \cup \{\$\}\})^*$. We conclude that $match(u|_S)_{|V} \in L(\mathcal{A}_V)$, by letting V select the dummy mode.

Assume now that u contains the action $V!G(\#)$. According to the definition of $\mathcal{A}_V$, this means that $u_{|V}$ is equal to $V?R(a_1)\, V?G(\overline{a_1}) \cdots V?R(a_n)\, V?G(\overline{a_n}) \cdot V?R(\$)\, V?G(\overline{\$})\, V!G(\#)$ for some $a_1, \ldots, a_n \in \Sigma$ with $n \geq 0$. The mailbox semantics entails that the word sent to V in u admits $a_1\overline{a_1} \cdots a_n\overline{a_n}\$\overline{\$}$ as a prefix. When we dealt about the relayer process, we showed that the word sent by R to V is in the language $\Sigma^* \cdot \{\varepsilon, \$\}$. Moreover, according to the definition of $\mathcal{A}_G$, the word sent by G to V is in the language $\overline{\Sigma}^* \cdot \{\varepsilon, \overline{\$}\}$. So the word sent to V in u is exactly $a_1\overline{a_1} \cdots a_n\overline{a_n}\$\overline{\$}$. We derive that the sequence of send actions to V in u is $v = R!V(a_1)\, G!V(\overline{a_1}) \cdots R!V(a_n)\, G!V(\overline{a_n}) \cdot R!V(\$)\, G!V(\overline{\$})$. Moreover, the single send action performed by V, namely $V!G(\#)$, is necessarily after $G!V(\overline{\$})$. This is due to the receive action $V?G(\overline{\$})$. It follows that $match(u|_S)_{|V}$ is equal to $V?R(a_1)\, V?G(\overline{a_1}) \cdots V?R(a_n)\, V?G(\overline{a_n}) \cdot V?R(\$)\, V?G(\overline{\$})\, V!G(\#)$, hence, $match(u|_S)_{|V} = u_{|V}$. We conclude that $match(u|_S)_{|V} \in L(\mathcal{A}_V)$. ◀

▶ **Corollary 3.5.** *If $\mathcal{I}$ is a negative instance of Pre-MPCP then $\mathcal{A}(\mathcal{I})$ is send-synchronizable.*

Proof. If $\mathcal{I}$ is a negative instance of Pre-MPCP then we derive from Lemma 3.3 that no trace in $Tr(\mathcal{A}(\mathcal{I}))$ contains the action $R!V(\#)$. It follows from Lemma 3.4 that $match(u|_S)_{|p} \in L(\mathcal{A}_p)$ for every trace $u \in Tr(\mathcal{A}(\mathcal{I}))$ and every process $p \in \{G, R, V\}$. We conclude thanks to Lemma B.1 that $\mathcal{A}(\mathcal{I})$ is send-synchronizable. ◀

Expectation in Stochastic Games with Prefix-Independent Objectives

Laurent Doyen
Université Paris-Saclay, CNRS, ENS Paris-Saclay, LMF, Gif-sur-Yvette, France

Pranshu Gaba
Tata Institute of Fundamental Research, Mumbai, India

Shibashis Guha
Tata Institute of Fundamental Research, Mumbai, India

Abstract

Stochastic two-player games model systems with an environment that is both adversarial and stochastic. In this paper, we study the expected value of bounded quantitative prefix-independent objectives in the context of stochastic games. We show a generic reduction from the expectation problem to linearly many instances of the almost-sure satisfaction problem for threshold Boolean objectives. The result follows from partitioning the vertices of the game into so-called value classes where each class consists of vertices of the same value. Our procedure further entails that the memory required by both players to play optimally for the expectation problem is no more than the memory required by the players to play optimally for the almost-sure satisfaction problem for a corresponding threshold Boolean objective.

We show the applicability of the framework to compute the expected window mean-payoff measure in stochastic games. The window mean-payoff measure strengthens the classical mean-payoff measure by computing the mean payoff over windows of bounded length that slide along an infinite path. We show that the decision problem to check if the expected window mean-payoff value is at least a given threshold is in $\mathsf{UP} \cap \mathsf{coUP}$ when the window length is given in unary.

2012 ACM Subject Classification Mathematics of computing → Stochastic processes; Theory of computation → Probabilistic computation

Keywords and phrases Stochastic games, finitary objectives, mean payoff, reactive synthesis

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2025.16

Related Version *Full Version*: https://arxiv.org/abs/2405.18048 [21]

1 Introduction

Reactive systems typically have an infinite execution where the controller continually reacts to the environment. Given a specification, the *reactive controller synthesis* problem [19] concerns with synthesising a policy for the controller such that the specification is satisfied by the system for all behaviours of the environment. This problem is modelled using two-player turn-based games on graphs, where the two players are the controller (Player 1) and the environment (Player 2), the vertices and the edges of the game graph represent the states and transitions of the system, and the objective of Player 1 is to satisfy the specification. An execution of the system is then an infinite path in the game graph. The reactive controller synthesis problem corresponds to determining if there exists a strategy of Player 1 such that for all strategies of Player 2, the outcome satisfies the objective. If such a winning strategy exists, then we would also like to synthesise it. The environment is considered as an adversarial player to ensure that the specification is met even in the worst-case scenario.

Objectives are either Boolean or quantitative. Each execution either satisfies a Boolean objective ψ or does not satisfy ψ. The set of executions that satisfy ψ form a language over infinite words with the alphabet being the set of vertices in the graphs. On the other hand,

© Laurent Doyen, Pranshu Gaba, and Shibashis Guha;
licensed under Creative Commons License CC-BY 4.0

a quantitative objective φ evaluates the performance of the execution by a numerical metric, which Player 1 aims to maximise and Player 2 aims to minimise. A quantitative objective can be viewed as a real-valued function over infinite paths in the graph.

In the presence of uncertainty or probabilistic behaviour, the game graph becomes stochastic. Fixing the strategies of the two players gives a distribution over infinite paths in the game graph. For Boolean objectives ψ, the goal of Player 1 is to maximise the probability that an outcome satisfies ψ. For quantitative objectives φ, there are two possible views:

Satisfaction. Given a threshold λ, to maximise the probability that φ-value of the outcome is greater than λ;

Expectation. To maximise the φ-value of the outcome in expectation.

Either view may be desirable depending on the context [5–8]. The satisfaction view can be seen as a Boolean objective: the φ-value of the outcome is either greater than λ or it is not. The expectation view is more nuanced, and is the subject of study in this paper.

In this paper, we look at the expectation problem for quantitative prefix-independent objectives (also referred to as tail objectives). These are objectives that do not depend on finite prefixes of the plays, but only on the long-run behaviour of the system. In systems, we are often willing to allow undesirable behaviour in the short-term, if the long run behaviour is desirable. Prefix-independent objectives model such requirements and thus are of interest to study [9]. Prefix-independent objectives also have the benefit that they satisfy the Bellman equations [34], which simplifies their analysis. The expectation problem for such objectives arises naturally in many scenarios. For example:

(i) An algorithmic trading system is designed to generate profit by executing trades based on real-time market data. Following an initial phase of learning and unstable behaviour due to parameter tuning, average profit over a bounded time window must always exceed a threshold and decisions need to be made within short well-defined intervals for them to be effective.

(ii) A power plant may have different strategies to produce power (such as coal, solar, nuclear, wind) and must allocate resources among these strategies so as to maximise the power produced in expectation.

Contributions. All of our contributions are with regard to quantitative prefix-independent objectives φ that are bounded (i.e., the image of φ is bounded between integers $-\mathsf{W}_\varphi$ and W_φ) and such that a bound den_φ on the denominators of the optimal expected φ-values of vertices in the game is known. The bound on the image ensures determinacy [35], that is, the players have optimal strategies, and the bound on the denominator of optimal values of vertices discretise the search space. These bounds often exist and are easily derivable for common objectives of interest such as mean payoff.

Our primary contribution is a reduction of the expectation problem for such an objective φ to linearly many instances of the almost-sure satisfaction problem for threshold Boolean objectives $\{\varphi > \lambda\}$ for thresholds $\lambda \in \mathbb{Q}$. Deciding the almost-sure satisfaction of $\{\varphi > \lambda\}$ is conceptually simpler than computing the expected value of φ, as in the former, we only need to consider if the measure of the paths that satisfy the objective $\{\varphi > \lambda\}$ is equal to one, whereas in the latter, one must take the averages of the measures of the sets of paths π weighted with the value $\varphi(\pi)$ of the paths. Our technique is generic in the sense that when an algorithm for the almost-sure satisfaction problem for $\{\varphi > \lambda\}$ is known, we directly obtain the complexity and a way to solve the expectation problem for φ.

Our reduction builds on the technique introduced in [16] for Boolean prefix-independent objectives and non-trivially extends it to quantitative prefix-independent objectives φ for which the bounds W_φ and den_φ are known. The expected φ-values of vertices are nondeterministic-

Table 1 Complexity and bounds on memory requirement for window mean-payoff objectives.

Objective	Complexity	Memory (Player 1) (lower [22], upper)	Memory (Player 2) (lower [22], upper)
FWMP(ℓ)	UP ∩ coUP	$\ell - 1$, ℓ	$\lvert V\rvert - \ell$, $\lvert V\rvert \cdot \ell$
BWMP	UP ∩ coUP	memoryless, memoryless	infinite, infinite

ally guessed, and we present a characterisation (Theorem 7, similar to [16, Lemma 8] but with important and subtle differences) to verify the guess. We also explicitly construct strategies for both players that are optimal for the expectation of φ, in terms of almost-sure winning strategies for $\{\varphi > \lambda\}$ (proof of Lemma 9). The memory requirement for the constructed optimal strategies is the same as that of the almost-sure winning strategies (Corollary 10).

Our framework gives an alternative approach to solve the expectation problem for well-studied objectives such as expected mean payoff and gives new results for not-as-well-studied objectives such as the *window mean-payoff objectives* introduced in [12]. As our secondary contribution, we illustrate our technique by applying it to two variants of window mean-payoff objectives: fixed (FWMP(ℓ)) and bounded (BWMP) window mean-payoff. Using our reduction, we are able to show that for both of these objectives, the expectation problem is in UP ∩ coUP (Theorem 18 and Theorem 22), a result that was not known before. The UP ∩ coUP upper bound for window mean-payoff objectives matches the special case of simple stochastic games [13, 20], and thus would require a major breakthrough to be improved. The lower bounds on the memory requirements for these objectives carry over from special case of the non-stochastic games [12, 22]. We summarise the complexity results and bounds on the memory requirements for the window mean-payoff objectives in Table 1.

Related work. Stochastic games were introduced by Shapley [38] where these games were studied under expectation semantics for discounted-sum objectives. In [14], it was shown that solving stochastic parity games reduces to solving stochastic mean-payoff games. Further, solving stochastic parity games, stochastic mean-payoff games, and *simple stochastic games* (i.e., stochastic games with reachability objective) are all polynomial-time equivalent [1, 30], and thus, are all in UP ∩ coUP [13]. A sub-exponential (or even quasi-polynomial) time deterministic algorithms for simple stochastic games on graphs with poly-logarithmic treewidth was proposed in [18]. In [29], sufficient conditions on the objective were shown such that optimal deterministic memoryless strategies exist for the players. In [34], value iteration to solve the expectation problem in stochastic games with reachability, safety, total-payoff, and mean-payoff objectives was studied.

Mean-payoff objectives were studied initially in two-player games, without stochasticity [24, 39], and with stochasticity in [28]. Finitary versions were introduced as window mean-payoff objectives [12]. For finitary mean-payoff objectives, the satisfaction problem [6] and the expectation problem [5] were studied in the special case of Markov decision processes (MDPs), which correspond to stochastic games with a trivial adversary. Expected mean payoff, expected discounted payoff, expected total payoff, etc., are widely studied for MDPs [4, 36]. Both the expectation problem [5] and the satisfaction problem [6] for the FWMP(ℓ) objective are in PTIME, while they are in UP ∩ coUP for the BWMP objective. Ensuring the satisfaction and expectation semantics simultaneously was studied in MDPs for the mean-payoff objective in [17] and for the window mean-payoff objectives in [26]. In both cases, the complexity was shown to be no greater than that of only expectation optimisation.

The satisfaction problem for window mean-payoff objectives has been studied for two-player stochastic games in [22]. While positive and almost-sure satisfaction of $\mathsf{FWMP}(\ell)$ are in PTIME, it follows from [22] that the problem is in $\mathsf{UP} \cap \mathsf{coUP}$ for quantitative satisfaction i.e., with threshold probabilities $0 < p < 1$. Furthermore, the satisfaction problem of BWMP is in $\mathsf{UP} \cap \mathsf{coUP}$ and thus has the same complexity as that of the special case of MDPs [6].

Due to lack of space, some of the proofs and details have been omitted. A full version of the paper can be found in [21].

2 Preliminaries

Probability distributions. A *probability distribution* over a finite non-empty set A is a function $\mathsf{Pr}\colon A \to [0,1]$ such that $\sum_{a \in A} \mathsf{Pr}(a) = 1$. We denote by $\mathcal{D}(A)$ the set of all probability distributions over A. For the algorithmic and complexity results, we assume that probabilities are given as rational numbers.

Stochastic games. We consider two-player turn-based zero-sum stochastic games (or simply, stochastic games in the sequel). The two players are referred to as Player 1 (she/her) and Player 2 (he/him). A *stochastic game* is given by $\mathcal{G} = ((V,E),(V_1,V_2,V_\Diamond),\mathbb{P},w)$, where:

- (V,E) is a directed graph with a finite set V of vertices and a set $E \subseteq V \times V$ of directed edges such that for all vertices $v \in V$, the set $E(v) = \{v' \in V \mid (v,v') \in E\}$ of out-neighbours of v is non-empty, i.e., $E(v) \neq \varnothing$ (no deadlocks).
- $(V_1,V_2,V_\Diamond)$ is a partition of V. The vertices in V_1 belong to Player 1, the vertices in V_2 belong to Player 2, and the vertices in $V_\Diamond$ are called *probabilistic vertices*;
- $\mathbb{P}\colon V_\Diamond \to \mathcal{D}(V)$ is a *probability function* that describes the behaviour of probabilistic vertices in the game. It maps each probabilistic vertex $v \in V_\Diamond$ to a probability distribution $\mathbb{P}(v)$ over the set $E(v)$ of out-neighbours of v such that $\mathbb{P}(v)(v') > 0$ for all $v' \in E(v)$ (i.e., all out-neighbours have non-zero probability);
- $w\colon E \to \mathbb{Q}$ is a *payoff function* assigning a rational payoff to every edge in the game.

Stochastic games are played in rounds. The game starts by initially placing a token on some vertex. At the beginning of a round, if the token is on a vertex v, and $v \in V_i$ for $i \in \{1,2\}$, then Player i chooses an out-neighbour $v' \in E(v)$; otherwise $v \in V_\Diamond$, and an out-neighbour $v' \in E(v)$ is chosen with probability $\mathbb{P}(v)(v')$. Player 1 receives from Player 2 the amount $w(v,v')$ given by the payoff function, and the token moves to v' for the next round. This continues ad infinitum resulting in an infinite sequence $\pi = v_0v_1v_2\cdots \in V^\omega$ such that $(v_i,v_{i+1}) \in E$ for all $i \geq 0$, called a *play*. For $i < j$, we denote by $\pi(i,j)$ the *infix* $v_iv_{i+1}\cdots v_j$ of π. Its length is $|\pi(i,j)| = j - i$, the number of edges. We denote by $\pi(0,j)$ the finite *prefix* $v_0v_1\cdots v_j$ of π, and by $\pi(i,\infty)$ the infinite *suffix* $v_iv_{i+1}\ldots$ of π. We denote by $\mathsf{Plays}_\mathcal{G}$ and $\mathsf{Prefs}_\mathcal{G}$ the set of all plays and the set of all finite prefixes in $\mathcal{G}$ respectively. We denote by $\mathsf{Last}(\rho)$ the last vertex of the prefix $\rho \in \mathsf{Prefs}_\mathcal{G}$. We denote by $\mathsf{Prefs}^i_\mathcal{G}$ $(i \in \{1,2\})$ the set of all prefixes ρ such that $\mathsf{Last}(\rho) \in V_i$.

A stochastic game with $V_\Diamond = \varnothing$ is a *non-stochastic two-player game*, a stochastic game with $V_2 = \varnothing$ is a *Markov decision process (MDP)*, and a stochastic game with $V_1 = V_2 = \varnothing$ is a *Markov chain*. Figure 1 shows an example of a stochastic game; Player 1 vertices are shown as circles, Player 2 vertices as boxes, and probabilistic vertices as diamonds.

Subgames and traps. Given a stochastic game $\mathcal{G} = ((V,E),(V_1,V_2,V_\Diamond),\mathbb{P},w)$, a subset $V' \subseteq V$ of vertices *induces* a subgame if (i) every vertex $v' \in V'$ has an outgoing edge in V', that is $E(v') \cap V' \neq \varnothing$, and (ii) every probabilistic vertex $v' \in V_\Diamond \cap V'$ has all outgoing edges

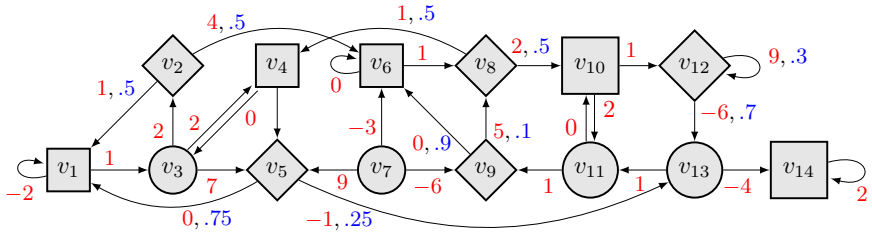


Figure 1 A stochastic game. Player 1 vertices are denoted by circles, Player 2 vertices are denoted by boxes, and probabilistic vertices are denoted by diamonds. The payoff for each edge is shown in red and probability distribution out of probabilistic vertices is shown in blue.

in V', that is $E(v') \subseteq V'$. The induced *subgame* is $((V', E'), (V_1 \cap V', V_2 \cap V', V_\Diamond \cap V'), \mathbb{P}', w')$, where $E' = E \cap (V' \times V')$, and $\mathbb{P}'$ and w' are restrictions of $\mathbb{P}$ and w respectively to (V', E'). If $T \subseteq V$ is such that for all $v \in T$, if $v \in V_1 \cup V_\Diamond$ then $E(v) \subseteq T$ and if $v \in V_2$ then $E(v) \cap T \neq \varnothing$, then T induces a subgame, and the subgame is a *trap* for Player 1 in $\mathcal{G}$, since Player 2 can ensure that if the token reaches T, then it never escapes.

Boolean objectives. A *Boolean objective* ψ is a Borel-measurable subset of $\mathsf{Plays}_\mathcal{G}$ [35]. A play $\pi \in \mathsf{Plays}_\mathcal{G}$ *satisfies* an objective ψ if $\pi \in \psi$. In a stochastic game $\mathcal{G}$ with objective ψ, the objective of Player 1 is ψ, and since $\mathcal{G}$ is a zero-sum game, the objective of Player 2 is the complement set $\overline{\psi} = \mathsf{Plays}_\mathcal{G} \setminus \psi$. An example of a Boolean objective is *reachability*, denoted $\mathsf{Reach}(T)$, the set of all plays that visit a vertex in the target set $T \subseteq V$. This is formally defined and more examples of Boolean objectives are given in [21].

Quantitative objectives. A *quantitative objective* is a Borel-measurable function of the form $\varphi \colon \mathsf{Plays}_\mathcal{G} \to \mathbb{R}$. In a stochastic game $\mathcal{G}$ with objective φ, the objective of Player 1 is φ and the objective of Player 2 is $-\varphi$, the negative of φ. Let $\pi = v_0 v_1 v_2 \cdots$ be a play. Some common examples of quantitative objectives include the *mean-payoff* objective $\mathsf{MP}(\pi) = \liminf_{n\to\infty} \frac{1}{n} \sum_{i=0}^{n} w(v_i, v_{i+1})$, and the *liminf* objective $\mathsf{liminf}(\pi) = \liminf_{n\to\infty} w(v_n, v_{n+1})$. In this work, we also consider the *window mean-payoff* objective, which is defined in Section 4. Corresponding to a quantitative objective φ, we define *threshold objectives* which are Boolean objectives ψ of the form $\{\pi \in \mathsf{Plays}_\mathcal{G} \mid \varphi(\pi) \bowtie \lambda\}$ for thresholds $\lambda \in \mathbb{R}$ and for $\bowtie \in \{<, \leq, >, \geq\}$. We denote this threshold objective succinctly as $\{\varphi \bowtie \lambda\}$.

Prefix independence. An objective is said to be *prefix-independent* if it only depends on the suffix of a play. Formally, a Boolean objective ψ is prefix-independent if for all plays π and π' with a common suffix (that is, π' can be obtained from π by removing and adding a finite prefix), we have that $\pi \in \psi$ if and only if $\pi' \in \psi$. Similarly, a quantitative objective φ is prefix-independent if for all plays π and π' with a common suffix, we have that $\varphi(\pi) = \varphi(\pi')$. Mean payoff and liminf are examples of prefix-independent objectives, whereas reachability and discounted payoff [2] are not.

Strategies. A (deterministic or pure) *strategy*[1] for Player $i \in \{1,2\}$ in a game $\mathcal{G}$ is a function $\sigma_i : \mathsf{Prefs}^i_{\mathcal{G}} \to V$ that maps prefixes ending in a vertex $v \in V_i$ to a successor of v. Strategies can be realised as the output of a (possibly infinite-state) Mealy machine [33]. The *memory size* of a strategy σ_i is the smallest number of states a Mealy machine defining σ_i can have. A strategy σ_i is *memoryless* if $\sigma_i(\rho)$ only depends on the last element of the prefix ρ, that is, for all prefixes $\rho, \rho' \in \mathsf{Prefs}^i_{\mathcal{G}}$ if $\mathsf{Last}(\rho) = \mathsf{Last}(\rho')$, then $\sigma_i(\rho) = \sigma_i(\rho')$.

A *strategy profile* $\sigma = (\sigma_1, \sigma_2)$ is a pair of strategies σ_1 and σ_2 of Player 1 and Player 2 respectively. A play $\pi = v_0 v_1 \cdots$ is *consistent* with a strategy σ_i of Player i ($i \in \{1,2\}$) if for all $j \geq 0$ with $v_j \in V_i$, we have $v_{j+1} = \sigma_i(\pi(0,j))$. A play π is an *outcome* of a profile $\sigma = (\sigma_1, \sigma_2)$ if it is consistent with both σ_1 and σ_2. For a Boolean objective ψ, we denote by $\mathsf{Pr}^{\sigma_1,\sigma_2}_{\mathcal{G},v}(\psi)$ the probability that an outcome of the profile (σ_1, σ_2) in $\mathcal{G}$ with initial vertex v satisfies ψ.

Satisfaction probability of Boolean objectives. Let ψ be a Boolean objective. A strategy σ_1 of Player 1 is *winning* with probability p from a vertex v in $\mathcal{G}$ for objective ψ if $\mathsf{Pr}^{\sigma_1,\sigma_2}_{\mathcal{G},v}(\psi) \geq p$ for all strategies σ_2 of Player 2. A strategy σ_1 of Player 1 is *positive* winning (resp., *almost-sure* winning) from v for Player 1 in $\mathcal{G}$ with objective ψ if $\mathsf{Pr}^{\sigma_1,\sigma_2}_{\mathcal{G},v}(\psi) > 0$ (resp., $\mathsf{Pr}^{\sigma_1,\sigma_2}_{\mathcal{G},v}(\psi) = 1$) for all strategies σ_2 of Player 2. In the above, if such a strategy σ_1 exists, then the vertex v is said to be positive winning (resp., almost-sure winning) for Player 1. If a vertex v is positive winning (resp., almost-sure winning) for Player 1, then Player 1 is said to play *optimally* from v if she follows a positive (resp., almost-sure) winning strategy from v. We omit analogous definitions for Player 2.

Expected value of quantitative objectives. Let φ be a quantitative objective. Given a strategy profile $\sigma = (\sigma_1, \sigma_2)$ and an initial vertex v, let $\mathbb{E}^{\sigma}_v(\varphi)$ denote the *expected φ-value of the outcome* of the strategy profile σ from v, that is, the expectation of φ over all plays with initial vertex v under the probability measure $\mathsf{Pr}^{\sigma_1,\sigma_2}_{\mathcal{G},v}(\varphi)$. We only consider objectives φ that are Borel-measurable and whose image is bounded. Thus, by determinacy of Blackwell games [35], we have that stochastic games with objective φ are determined. That is, we have $\sup_{\sigma_1} \inf_{\sigma_2} \mathbb{E}^{\sigma}_v(\varphi) = \inf_{\sigma_2} \sup_{\sigma_1} \mathbb{E}^{\sigma}_v(\varphi)$. We call this quantity the *expected φ-value of the vertex v* and denote it by $\mathbb{E}_v(\varphi)$. We say that Player 1 plays *optimally* from a vertex v if she follows a strategy σ_1 such that for all strategies σ_2 of Player 2, the expected φ-value of the outcome is at least $\mathbb{E}_v(\varphi)$. Similarly, Player 2 plays *optimally* if he follows a strategy σ_2 such that for all strategies σ_1 of Player 1, the expected φ-value of the outcome is at most $\mathbb{E}_v(\varphi)$. If φ is a prefix-independent objective, then we have the following relation between the expected φ-value of a vertex v and the expected φ-values of its out-neighbours.

▶ **Proposition 1** (*Bellman equations*). *If φ is a prefix-independent objective, then the following equations hold for all $v \in V$.*

$$\mathbb{E}_v(\varphi) = \begin{cases} \max_{v' \in E(v)} \mathbb{E}_{v'}(\varphi) & \text{if } v \in V_1 \\ \min_{v' \in E(v)} \mathbb{E}_{v'}(\varphi) & \text{if } v \in V_2 \\ \sum_{v' \in E(v)} \mathbb{P}(v)(v') \cdot \mathbb{E}_{v'}(\varphi) & \text{if } v \in V_\Diamond \end{cases}$$

[1] We only consider the satisfaction and expectation of Borel-measurable objectives, and deterministic strategies suffice for such objectives [10]. Satisfying two goals simultaneously, e.g., $\mathsf{Pr}(\mathsf{Reach}(T_1)) > 0.5 \wedge \mathsf{Pr}(\mathsf{Reach}(T_2)) > 0.5$ requires randomisation and is not allowed by our definition.

In this paper, we consider the expectation problem for prefix-independent objectives. Our solution in turn uses the almost-sure satisfaction problem. The decision problems are defined as follows.

Decision problems. Given a stochastic game $\mathcal{G}$, a quantitative objective φ, a vertex v, and a threshold $\lambda \in \mathbb{Q}$, the following decision problems are relevant:

- *almost-sure satisfaction problem*: Is vertex v almost-sure winning for Player 1 for a threshold objective $\{\varphi > \lambda\}$?
- *expectation problem*: Is $\mathbb{E}_v(\varphi) \geq \lambda$? That is, is the expected φ-value of v at least λ?

The reader is pointed to [2] and [25] for a more comprehensive discussion on the above-mentioned concepts.

3 Reducing expectation to almost-sure satisfaction

In this section, we show a reduction (Theorem 7) of the expectation problem for bounded quantitative prefix-independent objectives φ to the almost-sure satisfaction problem for the corresponding threshold objectives $\{\varphi > \lambda\}$. The reduction involves guessing a value r_v for every vertex v in the game, and then verifying if the guessed values are equal to the expected φ-values of the vertices. Theorem 7 generalises [16, Lemma 8] which studies the satisfaction problem for prefix-independent Boolean objectives, as Boolean objectives can be viewed as a special case of quantitative objectives by restricting the range to $\{0, 1\}$. We further discuss the difference in approaches between Theorem 7 and [16, Lemma 8] in Section 5.

Given a game $\mathcal{G}$ and a bounded prefix-independent quantitative objective φ, our reduction requires the existence of an integer bound den_φ on the denominators of expected φ-values of vertices in $\mathcal{G}$. Since φ is bounded, there exists an integer W_φ such that $|\varphi(\pi)| \leq \mathsf{W}_\varphi$ for every play π in $\mathcal{G}$. Thus, for every vertex v in $\mathcal{G}$, one can write $\mathbb{E}_v(\varphi)$ as $\frac{p}{q}$, where p and q are integers such that $|p| \leq \mathsf{W}_\varphi \cdot \mathsf{den}_\varphi$ and $0 < q \leq \mathsf{den}_\varphi$. The bounds W_φ and den_φ may depend on the objective and the structure of the graph, i.e., number of vertices, edge payoffs, probability distributions in the game, etc. These bounds effectively discretise the set of possible expected φ-values of the vertices, as there are at most $(2 \cdot \mathsf{W}_\varphi \cdot \mathsf{den}_\varphi + 1) \cdot \mathsf{den}_\varphi$ distinct possible values. This directly gives a bound on the granularity of the possible expected φ-values of vertices, that is, the minimum difference between two possible values of vertices, and we represent this quantity by ε_φ. Observe that given two rational numbers with denominators at most den_φ, the difference between them is at least $(\frac{1}{\mathsf{den}_\varphi})^2$, and thus, we let ε_φ be $(\frac{1}{\mathsf{den}_\varphi})^2$.

3.1 Value vectors and value classes

We first define and give notations for value vectors, which are useful in describing the reduction, and then look at some of their interesting and useful properties.

Definitions and notations. A *vector* $\vec{\mathsf{r}} = (\mathsf{r}_v)_{v \in V}$ of reals indexed by vertices in V induces a partition of V such that all vertices with the same value in $\vec{\mathsf{r}}$ belong to the same part in the partition. Let $k_{\vec{\mathsf{r}}}$ denote the number of parts in the partition, and let us denote the parts by $\{\mathsf{R}(1), \mathsf{R}(2), \ldots, \mathsf{R}(k_{\vec{\mathsf{r}}})\}$. We call each part $\mathsf{R}(i)$ of the partition an $\vec{\mathsf{r}}$-*class*, or simply, *class* if $\vec{\mathsf{r}}$ is clear from the context. For all $1 \leq i \leq k_{\vec{\mathsf{r}}}$, let $\mathfrak{r}_i$ denote the $\vec{\mathsf{r}}$-value of the class $\mathsf{R}(i)$. Given two vectors $\vec{\mathsf{r}}, \vec{\mathsf{s}}$, we write $\vec{\mathsf{r}} \geq \vec{\mathsf{s}}$ if for all $v \in V$, we have $\mathsf{r}_v \geq \mathsf{s}_v$, and we write $\vec{\mathsf{r}} > \vec{\mathsf{s}}$ if we have that $\vec{\mathsf{r}} \geq \vec{\mathsf{s}}$ and there exists $v \in V$ such that $\mathsf{r}_v > \mathsf{s}_v$. For a constant $c \in \mathbb{R}$, we denote by $\vec{\mathsf{r}} + c$ the vector obtained by adding c to each component of $\vec{\mathsf{r}}$.

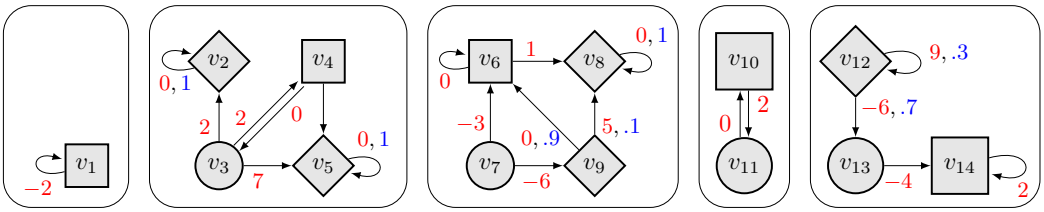


Figure 2 Restrictions $\mathcal{G}_{\mathsf{R}(1)}$, $\mathcal{G}_{\mathsf{R}(2)}$, $\mathcal{G}_{\mathsf{R}(3)}$, $\mathcal{G}_{\mathsf{R}(4)}$, and $\mathcal{G}_{\mathsf{R}(5)}$ of the game shown in Figure 1 for the vector $\vec{\mathsf{r}} = (-2, -1, -1, -1, -1, 0, 0, 0, 0, 1, 1, 2, 2, 2)$.

For all $1 \leq i \leq k_{\vec{\mathsf{r}}}$, a vertex $v \in \mathsf{R}(i)$ is a *boundary vertex* if v is a probabilistic vertex and has an out-neighbour not in $\mathsf{R}(i)$, i.e., if $v \in V_\Diamond$ and $E(v) \not\subseteq \mathsf{R}(i)$. Let $Bnd(\mathsf{R}(i))$ denote the set of boundary vertices in the class $\mathsf{R}(i)$. For all $1 \leq i \leq k_{\vec{\mathsf{r}}}$, let $\mathcal{G}_{\mathsf{R}(i)}$ denote the restriction of $\mathcal{G}$ to vertices in $\mathsf{R}(i)$ with all vertices in $Bnd(\mathsf{R}(i))$ changed to absorbing vertices with a self-loop. The edge payoffs of these self loops are not important (we assume them to be 0) as we restrict our attention to a subgame of $\mathcal{G}_{\mathsf{R}(i)}$ that does not contain boundary vertices.

▶ **Example 2.** For the game in Figure 1, let $\vec{\mathsf{r}} = (-2, -1, -1, -1, -1, 0, 0, 0, 0, 1, 1, 2, 2, 2)$ be a value vector for vertices $v_1, v_2, \ldots, v_{14}$ respectively. Since $\vec{\mathsf{r}}$ has five distinct values, we have $k_{\vec{\mathsf{r}}} = 5$, and the five $\vec{\mathsf{r}}$-classes are $\mathsf{R}(1) = \{v_1\}$, $\mathsf{R}(2) = \{v_2, v_3, v_4, v_5\}$, $\mathsf{R}(3) = \{v_6, v_7, v_8, v_9\}$, $\mathsf{R}(4) = \{v_{10}, v_{11}\}$, and $\mathsf{R}(5) = \{v_{12}, v_{13}, v_{14}\}$ with values $\mathfrak{r}_1 = -2$, $\mathfrak{r}_2 = -1$, $\mathfrak{r}_3 = 0$, $\mathfrak{r}_4 = 1$, and $\mathfrak{r}_5 = 2$ respectively. Out of the five probabilistic vertices v_2, v_5, v_8, v_9 and v_{12}, we see that v_2, v_5, and v_8 are boundary vertices while v_9 and v_{12} are not. Thus, $Bnd(\mathsf{R}(2)) = \{v_2, v_5\}$, $Bnd(\mathsf{R}(3)) = \{v_8\}$, and $Bnd(\mathsf{R}(1)) = Bnd(\mathsf{R}(4)) = Bnd(\mathsf{R}(5)) = \varnothing$. We show the restrictions $\mathcal{G}_{\mathsf{R}(i)}$ in Figure 2. ⌟

Let φ be a bounded prefix-independent quantitative objective. Analogous to the notation of a general value vector $\vec{\mathsf{r}}$, we describe notations for the expected φ-value vector consisting of the expected φ-values of vertices in V. For all vertices $v \in V$, let s_v denote $\mathbb{E}_v(\varphi)$, the expected φ-value of vertex v, and let $\vec{\mathsf{s}} = (\mathsf{s}_v)_{v \in V}$ denote the expected φ-value vector. Let $\mathsf{S}(i)$ denote the i^{th} $\vec{\mathsf{s}}$-class and let $\mathfrak{s}_i$ denote the $\vec{\mathsf{s}}$-value of $\mathsf{S}(i)$.

Given a vector $\vec{\mathsf{r}}$, it follows from Proposition 1 that the following is a necessary (but not sufficient) condition for $\vec{\mathsf{r}}$ to be the expected φ-value vector $\vec{\mathsf{s}}$.

- ***Bellman*** condition: for every vertex $v \in V$, the following Bellman equations hold

$$\mathsf{r}_v = \begin{cases} \max_{v' \in E(v)} \mathsf{r}_{v'} & \text{if } v \in V_1, \\ \min_{v' \in E(v)} \mathsf{r}_{v'} & \text{if } v \in V_2, \\ \sum_{v' \in E(v)} \mathbb{P}(v)(v') \cdot \mathsf{r}_{v'} & \text{if } v \in V_\Diamond. \end{cases}$$

Consequences of the Bellman condition. We now see some properties of value vectors $\vec{\mathsf{r}}$ that satisfy the **Bellman** condition. Since boundary vertices are probabilistic vertices, the following is immediate.

▶ **Proposition 3.** *Let $\vec{\mathsf{r}}$ be a value vector satisfying the **Bellman** condition. Then for all $1 \leq i \leq k_{\vec{\mathsf{r}}}$, for all $v \in Bnd(\mathsf{R}(i))$, there exists an out-neighbour of v with $\vec{\mathsf{r}}$-value less than $\mathfrak{r}_i$ and there exists an out-neighbour of v with $\vec{\mathsf{r}}$-value greater than $\mathfrak{r}_i$. Formally, there exist $1 \leq i_1, i_2 \leq k_{\vec{\mathsf{r}}}$ such that $\mathfrak{r}_{i_1} < \mathfrak{r}_i < \mathfrak{r}_{i_2}$ and $E(v) \cap \mathsf{R}(i_1) \neq \varnothing$ and $E(v) \cap \mathsf{R}(i_2) \neq \varnothing$.*

A corollary of Proposition 3 is that the $\vec{\mathsf{r}}$-classes with the smallest and the biggest $\vec{\mathsf{r}}$-values have no boundary vertices. Note that there may also exist $\vec{\mathsf{r}}$-classes other than these that do not contain boundary vertices (see $\mathsf{R}(4)$ in Example 2). Next, we see that the **Bellman** condition entails that each restriction $\mathcal{G}_{\mathsf{R}(i)}$ is a stochastic game.

▶ **Proposition 4.** *If $\vec{r}$ is a value vector that satisfies the **Bellman** condition, then for all $1 \leq i \leq k_{\vec{r}}$, we have that $\mathcal{G}_{\mathsf{R}(i)}$ is a stochastic game.*

In Proposition 5, we make a crucial observation about long-run behaviours of plays in $\mathcal{G}$, which is that either player can ensure with probability 1 that the token eventually reaches an $\vec{r}$-class from which it does not exit. This follows from the Borel-Cantelli lemma [23] due to the fact that there is a positive probability to reach an $\vec{r}$-class without boundary vertices following a finite number of edges out of boundary vertices.

▶ **Proposition 5.** *Let $\vec{r}$ be a value vector satisfying the **Bellman** condition. Suppose the strategy of Player i ($i \in \{1,2\}$) is such that each time the token reaches a vertex $v \in V_i$, (s)he moves the token to a vertex v' in the same $\vec{r}$-class as v. Then, with probability 1, the token eventually reaches a class $\mathsf{R}(j)$ for some $1 \leq j \leq k_{\vec{r}}$ from which it never exits.*

Finally, we define the notion of *trap subgames* of $\mathcal{G}_{\mathsf{R}(i)}$ which will be used in the subsequent discussion. We denote by $P^1_{\mathsf{R}(i)}$ the Player 1 positive attractor set of $Bnd(\mathsf{R}(i))$, i.e., the set of vertices in $\mathcal{G}_{\mathsf{R}(i)}$ that are positive winning for Player 1 for the $\mathsf{Reach}(Bnd(\mathsf{R}(i)))$ objective. The complement $T^1_{\mathsf{R}(i)} = \mathsf{R}(i) \setminus P^1_{\mathsf{R}(i)}$ is a trap for Player 1 in $\mathcal{G}_{\mathsf{R}(i)}$, and with abuse of notation, we use the same symbol $T^1_{\mathsf{R}(i)}$ to denote the subset of $\mathcal{G}_{\mathsf{R}(i)}$ as well as the trap subgame. We note that if $\mathsf{R}(i)$ does not have boundary vertices, that is, if $Bnd(\mathsf{R}(i)) = \varnothing$, then it holds that $P^1_{\mathsf{R}(i)} = \varnothing$ and $T^1_{\mathsf{R}(i)} = \mathsf{R}(i)$. We can analogously define $P^2_{\mathsf{R}(i)}$ and $T^2_{\mathsf{R}(i)}$ for Player 2. Given $\mathcal{G}_{\mathsf{R}(i)}$, these sets can be computed in polynomial time using attractor computations [25].

▶ **Example 6.** We compute these sets for the restrictions shown in Figure 2. For $i \in \{1,4,5\}$, since $Bnd(\mathsf{R}(i))$ is empty, we have that $T^1_{\mathsf{R}(i)} = \mathsf{R}(i)$ and $P^1_{\mathsf{R}(i)} = \varnothing$. For $\mathsf{R}(2)$, we have that $P^1_{\mathsf{R}(2)} = \mathsf{R}(2)$, and $T^1_{\mathsf{R}(2)} = \varnothing$. For $\mathsf{R}(3)$, we have that $T^1_{\mathsf{R}(3)} = \{v_6\}$, $P^1_{\mathsf{R}(3)} = \{v_7, v_8, v_9\}$. ⌟

3.2 Characterisation of the value vector

We describe in Theorem 7 a necessary and sufficient set of conditions for a given vector $\vec{r}$ to be equal to the expected φ-value vector $\vec{s}$. In addition to **Bellman**, Theorem 7 makes use of two more conditions, which we define before stating the theorem.

- ***lower-bound*** condition: for all $1 \leq i \leq k_{\vec{r}}$, Player 1 wins $\{\varphi > \mathfrak{r}_i - \varepsilon_\varphi\}$ almost surely in the trap subgame $T^1_{\mathsf{R}(i)}$ from all vertices in $T^1_{\mathsf{R}(i)}$.
- ***upper-bound*** condition: for all $1 \leq i \leq k_{\vec{r}}$, Player 2 wins $\{\varphi < \mathfrak{r}_i + \varepsilon_\varphi\}$ almost surely in the trap subgame $T^2_{\mathsf{R}(i)}$ from all vertices in $T^2_{\mathsf{R}(i)}$.

▶ **Theorem 7.** *The only vector $\vec{r}$, whose every component has denominator at most den_φ, that satisfies **Bellman**, **lower-bound**, and **upper-bound** is the expected φ-value vector $\vec{s}$.*

Proof. We show in Lemma 8 that $\vec{s}$ satisfies the three conditions. We show in Lemma 9 that if $\vec{r}$ is a vector that satisfies the three conditions, then $\vec{r}$ is less than ε_φ distance away from $\vec{s}$, that is, $\vec{s} - \varepsilon_\varphi < \vec{r} < \vec{s} + \varepsilon_\varphi$. In particular, if each component of $\vec{r}$ can be written as $\frac{p}{q}$, where p, q are both integers and q is at most den_φ, then it follows that $\vec{r}$ is equal to $\vec{s}$. ◀

In the rest of the section, we prove Lemmas 8 and 9 used in the proof of Theorem 7.

▶ **Lemma 8.** *The expected φ-value vector $\vec{s}$ satisfies the three conditions in Theorem 7.*

Proof. The fact that $\vec{s}$ satisfies **Bellman** follows directly from Proposition 1. We show that **lower-bound** holds for $\vec{s}$. The proof for **upper-bound** is analogous.

Suppose for the sake of contradiction that **lower-bound** does not hold, that is, there exists $1 \leq i \leq k_{\vec{s}}$ and a vertex v in $T^1_{\mathsf{S}(i)}$ such that Player 2 has a positive winning strategy from v for the $\{\varphi \leq \mathfrak{s}_i - \varepsilon_\varphi\}$ objective in $T^1_{\mathsf{S}(i)}$. Since $\{\varphi \leq \mathfrak{s}_i - \varepsilon_\varphi\}$ is a prefix-independent objective, from [9, Theorem 1] we have that there exists another vertex v' in $T^1_{\mathsf{S}(i)}$ such that Player 2 has an almost-sure winning strategy from v' for the same objective $\{\varphi \leq \mathfrak{s}_i - \varepsilon_\varphi\}$ in $T^1_{\mathsf{S}(i)}$. If Player 2 follows this strategy from v' in the *original game* $\mathcal{G}$, then one of the following two cases holds

- Player 1 always moves the token to a vertex in $\mathsf{S}(i)$. Since v' is in the trap $T^1_{\mathsf{S}(i)}$ for Player 1 in $\mathcal{G}_{\mathsf{S}(i)}$, Player 2 can force the token to remain in $T^1_{\mathsf{S}(i)}$ forever, and follow the almost-sure winning strategy to ensure that with probability 1, the outcome satisfies the objective $\{\varphi \leq \mathfrak{s}_i - \varepsilon_\varphi\}$.
- Player 1 eventually moves the token to a vertex out of $\mathsf{S}(i)$. Since $\vec{\mathsf{s}}$ satisfies **Bellman**, the token moves to an $\vec{\mathsf{s}}$-class with a smaller value than $\mathfrak{s}_i$.

In both cases, the expected φ-value of the outcome is less than $\mathfrak{s}_i$. This is a contradiction since $v' \in \mathsf{S}(i)$, and the expected φ-value of every vertex in $\mathsf{S}(i)$ is equal to $\mathfrak{s}_i$. ◀

▶ **Lemma 9.** *If a vector $\vec{\mathsf{r}}$ satisfies the three conditions in Theorem 7, then $\vec{\mathsf{s}} - \varepsilon_\varphi < \vec{\mathsf{r}} < \vec{\mathsf{s}} + \varepsilon_\varphi$. In particular, we have the following:*

- *If $\vec{\mathsf{r}}$ satisfies the **Bellman** and **lower-bound** conditions, then $\vec{\mathsf{s}} > \vec{\mathsf{r}} - \varepsilon_\varphi$.*
- *If $\vec{\mathsf{r}}$ satisfies the **Bellman** and **upper-bound** conditions, then $\vec{\mathsf{s}} < \vec{\mathsf{r}} + \varepsilon_\varphi$.*

Proof sketch. We prove the first case. The proof for the second case follows by symmetry, that is, we essentially replace Player 1 by Player 2, and $\{\varphi > \mathfrak{r}_i - \varepsilon_\varphi\}$ by $\{\varphi < \mathfrak{r}_i + \varepsilon_\varphi\}$. We describe an optimal strategy σ^*_1 of Player 1 and give a sketch of its optimality.

Since $\vec{\mathsf{r}}$ satisfies the **lower-bound** condition, we have that for all $1 \leq i \leq k_{\vec{\mathsf{r}}}$, Player 1 has an an almost-sure winning strategy $\sigma^T_{\mathsf{R}(i)}$ in the trap subgame $T^1_{\mathsf{R}(i)}$ to win the objective $\{\varphi > \mathfrak{r}_i - \varepsilon_\varphi\}$ in $T^1_{\mathsf{R}(i)}$ almost surely from all vertices in $T^1_{\mathsf{R}(i)}$. From the definition of $P^1_{\mathsf{R}(i)}$, Player 1 has a positive winning strategy $\sigma^P_{\mathsf{R}(i)}$ in the restricted game $\mathcal{G}_{\mathsf{R}(i)}$ from vertices in $P^1_{\mathsf{R}(i)}$ for the $\mathsf{Reach}(Bnd(\mathsf{R}(i)))$ objective. By following $\sigma^P_{\mathsf{R}(i)}$, the token either reaches $Bnd(\mathsf{R}(i))$ with positive probability, or ends up in $T^1_{\mathsf{R}(i)}$ from where Player 2 can ensure that the token never leaves $T^1_{\mathsf{R}(i)}$. Using these strategies of Player 1 in $\mathcal{G}_{\mathsf{R}(i)}$, we construct a strategy σ^*_1 of Player 1 that is optimal for expected φ-value in the *original game* $\mathcal{G}$: As long as the token is in the class $\mathsf{R}(i)$ in $\mathcal{G}$, the strategy σ^*_1 mimics $\sigma^T_{\mathsf{R}(i)}$ if the token is in $T^1_{\mathsf{R}(i)}$ and σ^*_1 mimics $\sigma^P_{\mathsf{R}(i)}$ if the token is in $P^1_{\mathsf{R}(i)}$.

Note that whenever the token is on a vertex $v \in V_1$ in $\mathsf{R}(i)$, the strategy σ^*_1 always moves the token to a vertex v' in same $\vec{\mathsf{r}}$-class $\mathsf{R}(i)$ as v (i.e. a token only exits an $\vec{\mathsf{r}}$-class from a Player 2 vertex or from a boundary vertex), and thus, Proposition 5 holds. Whenever the token exits a class $\mathsf{R}(i)$ to reach a different class $\mathsf{R}(i')$, then as long as the token remains in $\mathsf{R}(i')$, the strategy σ^*_1 follows $\sigma^T_{\mathsf{R}(i')}$ if the token is in $T^1_{\mathsf{R}(i')}$, and σ^*_1 follows $\sigma^P_{\mathsf{R}(i')}$ if the token is in $P^1_{\mathsf{R}(i')}$.

Since Proposition 5 holds, we have that for all strategies of Player 2, with probability 1, the token eventually reaches an $\vec{\mathsf{r}}$-class $\mathsf{R}(j)$ from which it never exits. Moreover, the strategy σ^*_1 ensures that with probability 1, the token eventually reaches $T^1_{\mathsf{R}(j)}$ in $\mathsf{R}(j)$ from which it never leaves. Because if not, then the token would visit vertices in $P^1_{\mathsf{R}(j)}$ infinitely often, having a fixed positive probability of reaching $Bnd(\mathsf{R}(j))$ in every step because of $\sigma^P_{\mathsf{R}(j)}$. Thus, with probability 1, the token would eventually reach $Bnd(\mathsf{R}(i))$ from where it could escape to a different $\vec{\mathsf{r}}$-class, which contradicts the fact that the token stays in $\mathsf{R}(j)$ forever.

Since φ is prefix-independent, the φ-value of a play only depends on the trap $T^1_{\mathsf{R}(j)}$ it ends up in. If the game begins from a vertex $v \in \mathsf{R}(i)$, then for $1 \leq j \leq k_{\vec{\mathsf{r}}}$, let p_j denote the probability that the token ends up in the trap subgame $T^1_{\mathsf{R}(j)}$ from which it never exits. Since $\vec{\mathsf{r}}$ satisfies **Bellman**, we have that $\sum_j p_j \mathfrak{r}_j = \mathfrak{r}_i$. Since $\vec{\mathsf{r}}$ satisfies **lower-bound**, Player 1 has an almost-sure winning strategy for $\{\varphi > \mathfrak{r}_j - \varepsilon_\varphi\}$ in $T^1_{\mathsf{R}(j)}$. Thus, for all strategies σ_2 of Player 2, the expected value of an outcome of (σ_1^*, σ_2) from $v \in \mathsf{R}(i)$ is greater than $\sum_j p_j(\mathfrak{r}_j - \varepsilon_\varphi)$, which is $\mathfrak{r}_i - \varepsilon_\varphi$. This holds for all vertices v in $\mathcal{G}$, giving us the desired result $\vec{\mathsf{s}} > \vec{\mathsf{r}} - \varepsilon_\varphi$. ◀

We also note that the optimal strategy σ_1^* always either follows an almost-sure winning strategy $\sigma^T_{\mathsf{R}(i)}$ for the threshold objective $\{\varphi > \mathfrak{r}_i - \varepsilon_\varphi\}$ or a positive winning strategy for a Reach objective. Since there exist memoryless positive winning strategies for the Reach objective [20], we have the following bound on the memory requirement of σ_1^*.

▶ **Corollary 10.** *The memory requirement of σ_1^* is at most the maximum over all $1 \leq i \leq k_{\vec{\mathsf{r}}}$ of the memory requirement of an almost-sure winning strategy $\sigma^T_{\mathsf{R}(i)}$ for the threshold objective $\{\varphi > \mathfrak{r}_i - \varepsilon_\varphi\}$. Moreover, if $\sigma^T_{\mathsf{R}(i)}$ is a deterministic strategy, then so is σ_1^*.*

3.3 Bounding the denominators in the value vector

In this section, we discuss the problem of obtaining an upper bound den_φ for the denominators of the expected φ-values of vertices $\mathfrak{s}_i$ for a bounded prefix-independent objective φ in a game $\mathcal{G}$. In [16], the technique of value class is used to compute the values of vertices for Boolean prefix-independent objectives. It is stated without proof that the probability of satisfaction of a parity or a Streett objective [2] from each vertex can be written as $\frac{p}{q}$ where $q \leq (\widehat{\mathbb{P}})^{4\cdot|E|}$ and $\widehat{\mathbb{P}}$ is the maximum denominator over all edge probabilities in the game. As such, we were not able to directly generalise this bound for the expectation of quantitative prefix-independent objectives. Instead, we make the following observations:

- Let $\mathsf{S}(i)$ be an $\vec{\mathsf{s}}$-class without boundary vertices. If the token is in $\mathsf{S}(i)$ at some point in the play, then since $\vec{\mathsf{s}}$ satisfies the **Bellman** condition, neither player has an incentive to move the token out of $\mathsf{S}(i)$. Since there are no boundary vertices in $\mathsf{S}(i)$, the token does not exit $\mathsf{S}(i)$ from a probabilistic vertex either, and remains in $\mathsf{S}(i)$ forever. Thus, the value $\mathfrak{s}_i$ of $\mathsf{S}(i)$ depends only on the internal structure of $\mathsf{S}(i)$. We denote by $\underline{\mathsf{den}}_\varphi$ an upper bound on the denominators of values of $\vec{\mathsf{s}}$-classes without boundary vertices. It is a simpler problem to find $\underline{\mathsf{den}}_\varphi$ than to find den_φ, as each class without boundary vertices can be treated as a subgame in which each vertex has the same expected φ-value, or equivalently, the subgame consists of exactly one $\vec{\mathsf{s}}$-class.
- On the other hand, suppose $\mathsf{S}(i)$ is an $\vec{\mathsf{s}}$-class containing at least one boundary vertex, and let v be a boundary vertex in $\mathsf{S}(i)$. Then, since $\vec{\mathsf{s}}$ satisfies the **Bellman** condition, we have $\mathsf{s}_v = \sum_{v' \in E(v)} \mathbb{P}(v)(v') \cdot \mathsf{s}_{v'}$, which is also the value $\mathfrak{s}_i$ of $\mathsf{S}(i)$. Thus, $\mathfrak{s}_i$ can be written in terms of the values of classes reachable from v in one step and the probabilities with which those classes are reached. In fact, we construct in the proof of Theorem 11 a system of linear equations to show that the value of each $\vec{\mathsf{s}}$-class with boundary vertices can be expressed solely in terms of transition probabilities of the outgoing edges from boundary vertices and values of $\vec{\mathsf{s}}$-classes without boundary vertices.

The method to calculate $\underline{\mathsf{den}}_\varphi$ depends on the specific objective; we illustrate as an example in Section 4 a way to obtain $\underline{\mathsf{den}}_\varphi$ for a particular kind of objective called the window mean-payoff objective. Once we know $\underline{\mathsf{den}}_\varphi$ for an objective φ, we can use Theorem 11 to obtain den_φ in terms of $\underline{\mathsf{den}}_\varphi$.

▶ **Theorem 11.** *The denominator of the value of each $\vec{\mathsf{s}}$-class in $\mathcal{G}$ is at most $\mathsf{den}_\varphi = 2^{|V|} \cdot \widehat{\mathbb{P}}^{|V|^3} \cdot (\underline{\mathsf{den}}_\varphi)^{|V|}$.*

We note that this theorem implies that the number of bits required to write den_φ is polynomial in the number of vertices in the game and in the number of bits required to write $\underline{\mathsf{den}}_\varphi$. We devote the rest of this section to the proof of Theorem 11. For ease of notation, we denote the number of $\vec{\mathsf{s}}$-classes in the game by k instead of $k_{\vec{\mathsf{s}}}$ for the rest of this section. If every $\vec{\mathsf{s}}$-class has no boundary vertices, then we have den_φ equal to $\underline{\mathsf{den}}_\varphi$ and we are done. So we assume there exists at least one class that contains boundary vertices. Let $m \geq 1$ denote the number of $\vec{\mathsf{s}}$-classes with boundary vertices, and therefore, there are $k - m$ $\vec{\mathsf{s}}$-classes without boundary vertices. Since there always exists at least one $\vec{\mathsf{s}}$-class without boundary vertices (Proposition 3), we have that $m < k$. Let $B = \{1, 2, \ldots, m\}$ and $C = \{m+1, \ldots, k\}$. We index the $\vec{\mathsf{s}}$-classes such that each class with boundary vertices has its index in B and each class without boundary vertices has its index in C. Furthermore, in the sets B and C, the classes are indexed in increasing order of their values. That is, for i, j both in B or both in C, we have $i < j$ if and only if $\mathfrak{s}_i < \mathfrak{s}_j$. We show bounds on the denominators of $\vec{\mathsf{s}}$-values of classes with boundary vertices, i.e., $\mathfrak{s}_1, \ldots, \mathfrak{s}_m$ in terms of $\vec{\mathsf{s}}$-values of classes without boundary vertices, i.e., $\mathfrak{s}_{m+1}, \ldots, \mathfrak{s}_k$.

For all $i \in B = \{1, 2, \ldots, m\}$, pick an arbitrary boundary vertex from $Bnd(\mathsf{S}(i))$ and call this the *representative vertex* u_i of $Bnd(\mathsf{S}(i))$. For all $i \in B$ and $j \in \{1, 2, \ldots, k\}$, let $p_{i,j}$ denote the probability of reaching the class $\mathsf{S}(j)$ from u_i in one step. Since $\vec{\mathsf{s}}$ satisfies the **Bellman** condition, we have that $\sum_{1 \leq j \leq k} p_{i,j} \cdot \mathfrak{s}_j = \mathfrak{s}_i$. It is helpful to split this sum based on whether $j \in B$ or $j \in C$, i.e., whether $1 \leq j \leq m$ or $m+1 \leq j \leq k$. We rewrite the sums as $\sum_{j \in B} p_{i,j}\mathfrak{s}_j + \sum_{j \in C} p_{i,j}\mathfrak{s}_j = \mathfrak{s}_i$ for all $i \in B$, and we represent this system of equations below using matrices.

$$\begin{pmatrix} p_{1,1} & p_{1,2} & \cdots & p_{1,m} \\ p_{2,1} & p_{2,2} & \cdots & p_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ p_{m,1} & p_{m,2} & \cdots & p_{m,m} \end{pmatrix} \begin{pmatrix} \mathfrak{s}_1 \\ \mathfrak{s}_2 \\ \vdots \\ \mathfrak{s}_m \end{pmatrix} + \begin{pmatrix} p_{1,m+1} & p_{1,m+2} & \cdots & p_{1,k} \\ p_{2,m+1} & p_{2,m+2} & \cdots & p_{2,k} \\ \vdots & \vdots & \ddots & \vdots \\ p_{m,m+1} & p_{m,m+2} & \cdots & p_{m,k} \end{pmatrix} \begin{pmatrix} \mathfrak{s}_{m+1} \\ \mathfrak{s}_{m+2} \\ \vdots \\ \mathfrak{s}_k \end{pmatrix} = \begin{pmatrix} \mathfrak{s}_1 \\ \mathfrak{s}_2 \\ \vdots \\ \mathfrak{s}_m \end{pmatrix}$$

This system of equations is of the form $Q_B\mathfrak{s}_B + Q_C\mathfrak{s}_C = \mathfrak{s}_B$. Rearranging terms gives us $(I - Q_B)\mathfrak{s}_B = Q_C\mathfrak{s}_C$ where I is the $m \times m$ identity matrix. It follows from Proposition 12 that the equation $(I - Q_B)\mathfrak{s}_B = Q_C\mathfrak{s}_C$ has a unique solution.

▶ **Proposition 12.** *The matrix $(I - Q_B)$ is invertible.*

Let α denote the least common multiple (lcm) of the denominators of $p_{i,j}$ for $1 \leq i \leq m$ and $1 \leq j \leq k$. We have $0 < \alpha \leq \widehat{\mathbb{P}}^{mk}$, where $\widehat{\mathbb{P}}$ is the maximum denominator over all edge probabilities in $\mathcal{G}$. We multiply both sides of the equation $(I - Q_B)\mathfrak{s}_B = Q_c\mathfrak{s}_C$ by α to get $\alpha(I - Q_B)\mathfrak{s}_B = \alpha Q_C\mathfrak{s}_C$ and note that all the elements of $\alpha(I - Q_B)$ and αQ_C are integers. Let D be the determinant of the matrix $\alpha(I - Q_B)$, and for $1 \leq i \leq m$, let N_i be the determinant of the matrix obtained by replacing the i^{th} column of $\alpha(I - Q_B)$ with the column vector $\alpha Q_C\mathfrak{s}_C$. Since $\alpha(I - Q_B)$ is invertible, by Cramer's rule [32], we have that $\mathfrak{s}_i = N_i/D$. Proposition 13 shows that $|D|$ is an integer and is at most $(2\alpha)^m$ and Proposition 14 shows that N_i has denominator at most $(\underline{\mathsf{den}}_\varphi)^{k-m}$.

▶ **Proposition 13.** *The absolute value of the determinant of $\alpha(I - Q_B)$, i.e., $|D|$, is an integer and is at most $(2\alpha)^m$, which is at most $2^{|V|} \cdot \widehat{\mathbb{P}}^{|V|^3}$.*

▶ **Proposition 14.** *The denominator of N_i is at most $(\underline{\mathsf{den}}_\varphi)^{k-m}$, which is at most $(\underline{\mathsf{den}}_\varphi)^{|V|}$.*

Since the denominator of $\mathfrak{s}_i$ is at most $|D|$ times the denominator of N_i, we obtain the bound stated in Theorem 11. ◀

4 Expectation of window mean-payoff objectives

In this section, we apply the results from the previous section for two types of window mean-payoff objectives introduced in [12]:

(i) *fixed window mean-payoff* (FWMP(ℓ)) in which a window length $\ell \geq 1$ is given, and

(ii) *bounded window mean-payoff* (BWMP) in which for every play, we need a bound on window lengths.

We define these objectives below.

For a play π in a stochastic game $\mathcal{G}$, the *mean payoff* of an infix $\pi(i, i+n)$ is the average of the payoffs of the edges in the infix and is defined as $\mathsf{MP}(\pi(i, i+n)) = \sum_{k=i}^{i+n-1} \frac{1}{n} w(v_k, v_{k+1})$. Given a window length $\ell \geq 1$ and a threshold $\lambda \in \mathbb{R}$, a play $\pi = v_0 v_1 \cdots$ in $\mathcal{G}$ satisfies the *fixed window mean-payoff objective* $\mathsf{FWMP}_{\mathcal{G}}(\ell, \lambda)$ if from every position after some point, it is possible to start an infix of length at most ℓ with mean payoff at least λ.

$$\mathsf{FWMP}_{\mathcal{G}}(\ell, \lambda) = \{\pi \in \mathsf{Plays}_{\mathcal{G}} \mid \exists k \geq 0 \cdot \forall i \geq k \cdot \exists j \in \{1, \ldots, \ell\} \colon \mathsf{MP}(\pi(i, i+j)) \geq \lambda\}$$

We omit the subscript $\mathcal{G}$ when it is clear from the context. We extend the definition of *windows* as defined in [12] for arbitrary thresholds. Given a threshold λ, a play $\pi = v_0 v_1 \cdots$, and $0 \leq i < j$, we say that the λ-*window* $\pi(i, j)$ is *open* if the mean payoff of $\pi(i, k)$ is less than λ for all $i < k \leq j$. Otherwise, the λ-window is *closed*. A play π satisfies $\mathsf{FWMP}(\ell, \lambda)$ if and only if from some point on, every λ-window in π closes within at most ℓ steps. Note that $\mathsf{FWMP}(\ell, \lambda) \subseteq \mathsf{FWMP}(\ell', \lambda)$ for $\ell \leq \ell'$ as a smaller window length is a stronger constraint.

We also consider another window mean-payoff objective called the *bounded window mean-payoff objective* $\mathsf{BWMP}_{\mathcal{G}}(\lambda)$. A play satisfies the objective $\mathsf{BWMP}(\lambda)$ if there exists a window length $\ell \geq 1$ such that the play satisfies $\mathsf{FWMP}(\ell, \lambda)$.

$$\mathsf{BWMP}_{\mathcal{G}}(\lambda) = \{\pi \in \mathsf{Plays}_{\mathcal{G}} \mid \exists \ell \geq 1 : \pi \in \mathsf{FWMP}_{\mathcal{G}}(\ell, \lambda)\}$$

Note that both $\mathsf{FWMP}(\ell, \lambda)$ and $\mathsf{BWMP}(\lambda)$ are Boolean prefix-independent objectives.

Expected window mean-payoff values. Corresponding to the Boolean objectives $\mathsf{FWMP}(\ell, \lambda)$ and $\mathsf{BWMP}(\lambda)$, we define quantitative versions of these objectives. Given a play π in a stochastic game $\mathcal{G}$ and a window length ℓ, the $\varphi_{\mathsf{FWMP}(\ell)}$-value of π is $\sup\{\lambda \in \mathbb{R} \mid \pi \in \mathsf{FWMP}_{\mathcal{G}}(\ell, \lambda)\}$, the supremum threshold λ such that the play satisfies $\mathsf{FWMP}_{\mathcal{G}}(\ell, \lambda)$. Using notations from Section 2, we denote the expected $\varphi_{\mathsf{FWMP}(\ell)}$-value of a vertex v by $\mathbb{E}_v(\varphi_{\mathsf{FWMP}(\ell)})$. We define $\mathbb{E}_v(\varphi_{\mathsf{BWMP}})$, the expected φ_{BWMP}-value of a vertex v analogously. If W is an integer such that the payoff $w(e)$ of each edge e in $\mathcal{G}$ satisfies $|w(e)| \leq \mathsf{W}$, then for all plays π in $\mathcal{G}$, we have that $\varphi_{\mathsf{FWMP}(\ell)}(\pi)$ and $\varphi_{\mathsf{BWMP}}(\pi)$ lie between $-\mathsf{W}$ and W. Thus, $\varphi_{\mathsf{FWMP}(\ell)}$ and φ_{BWMP} are bounded objectives.

Decision problems. Given a stochastic game $\mathcal{G}$, a vertex v, and a threshold $\lambda \in \mathbb{Q}$, we have the following expectation problems for the window mean-payoff objectives:

- *expected* $\varphi_{FWMP(\ell)}$*-value problem*: Given a window length $\ell \geq 1$, is $\mathbb{E}_v(\varphi_{\mathsf{FWMP}(\ell)}) \geq \lambda$?
- *expected* φ_{BWMP}*-value problem*: Is $\mathbb{E}_v(\varphi_{\mathsf{BWMP}}) \geq \lambda$?

As considered in previous works [5, 6, 12], the window length ℓ is usually small ($\ell \leq |V|$), and hence we assume that ℓ is given in unary (while the edge-payoffs are given in binary).

4.1 Expected fixed window mean-payoff value

We give tight complexity bounds for the expected $\varphi_{\mathsf{FWMP}(\ell)}$-value problem. We use the characterisation from Theorem 7 to present our main result that this problem is in $\mathsf{UP} \cap \mathsf{coUP}$ (Theorem 18). We show in [21] that simple stochastic games [20], which are in $\mathsf{UP} \cap \mathsf{coUP}$ [13], reduce to the expected $\varphi_{\mathsf{FWMP}(\ell)}$-value problem, giving a tight lower bound.

In order to use the characterisation, we show the existence of the bound $\underline{\mathsf{den}}_{\mathsf{FWMP}(\ell)}$ for the $\varphi_{\mathsf{FWMP}(\ell)}$ objective. We show in Lemma 15 that the expected $\varphi_{\mathsf{FWMP}(\ell)}$-value $\mathfrak{s}_i$ of a class $\mathsf{S}(i)$ without boundary vertices takes a special form, that is, $\mathfrak{s}_i$ is the mean-payoff of a sequence of at most ℓ edges in $\mathsf{S}(i)$. We use the fact that the $\varphi_{\mathsf{FWMP}(\ell)}$-value of every play π is the largest λ such that, eventually, every λ-window in π closes in at most ℓ steps, and that λ is the mean payoff of a sequence of at most ℓ edges in π. To complete the argument, we show that if both players play optimally, then, with probability 1, the $\varphi_{\mathsf{FWMP}(\ell)}$-value of the outcome π is equal to $\mathfrak{s}_i$ and thus, $\mathfrak{s}_i$ is also of this form.

▶ **Lemma 15.** *The expected $\varphi_{\mathsf{FWMP}(\ell)}$-value $\mathfrak{s}_i$ of vertices in a class $\mathsf{S}(i)$ without boundary vertices is equal to the mean payoff of some sequence of ℓ or fewer edges in $\mathsf{S}(i)$. That is, $\mathfrak{s}_i$ is of the form $\frac{1}{j}\left(w(e_1) + \cdots + w(e_j)\right)$ for some $j \leq \ell$ and edges $e_1, e_2, \ldots, e_j$.*

This observation gives us the bound $\underline{\mathsf{den}}_{\mathsf{FWMP}(\ell)}$ on the denominators of the values of $\vec{\mathsf{s}}$-classes without boundary vertices. To see this, let $\widehat{w} = \max\{q \mid \exists p, q \in \mathbb{Z}, \exists e \in E : w(e) = \frac{p}{q} \text{ with } p, q \text{ co-prime}\}$ be the maximum denominator over all edge-payoffs in $\mathcal{G}$. Since $j \leq \ell$, and each $w(e_1), w(e_2), \ldots, w(e_j)$ is a rational number with denominator at most $\widehat{w}$, the denominator of the sum $w(e_1) + \cdots + w(e_j)$ is at most $\widehat{w} \cdot (\widehat{w} - 1) \cdot (\widehat{w} - 2) \cdots (\widehat{w} - (\ell - 1))$ if $\widehat{w} \geq \ell$, and at most $\widehat{w}!$ if $\widehat{w} \leq \ell$. In both cases, this is at most $\widehat{w}^\ell$.

▶ **Corollary 16.** *The expected $\varphi_{\mathsf{FWMP}(\ell)}$-value of vertices in $\vec{\mathsf{s}}$-classes without boundary vertices can be written as $\frac{p}{q}$ where p and q are integers and $q \leq \widehat{w}^\ell \cdot \ell$.*

From Theorem 11, we get that the denominator of $\mathfrak{s}_i$ for each class $\mathsf{S}(i)$ in $\mathcal{G}$ is at most $2^{|V|} \cdot \widehat{\mathbb{P}}^{|V|^3} \cdot (\underline{\mathsf{den}}_{\mathsf{FWMP}(\ell)})^{|V|}$, which is at most $2^{|V|} \cdot \widehat{\mathbb{P}}^{|V|^3} \cdot (\widehat{w}^\ell \cdot \ell)^{|V|}$.

▶ **Lemma 17.** *The expected $\varphi_{\mathsf{FWMP}(\ell)}$-value of each vertex in $\mathcal{G}$ can be written as a fraction $\frac{p}{q}$, where p, q are integers, and $q \leq 2^{|V|} \cdot \widehat{\mathbb{P}}^{|V|^3} \cdot (\widehat{w}^\ell \cdot \ell)^{|V|}$, and $-W \cdot q \leq p \leq W \cdot q$.*

We now state the main result of this section for the expected $\varphi_{\mathsf{FWMP}(\ell)}$-value problem.

▶ **Theorem 18.** *The expected $\varphi_{\mathsf{FWMP}(\ell)}$-value problem is in $\mathsf{UP} \cap \mathsf{coUP}$ when ℓ is given in unary. Memory of size ℓ suffices for Player 1, while memory of size $|V| \cdot \ell$ suffices for Player 2.*

Proof. To show membership of the expected $\varphi_{\mathsf{FWMP}(\ell)}$-value problem in $\mathsf{UP} \cap \mathsf{coUP}$, we first guess the expected $\varphi_{\mathsf{FWMP}(\ell)}$-value vector $\vec{\mathsf{s}}$, that is, the expected $\varphi_{\mathsf{FWMP}(\ell)}$-value s_v of every vertex v in the game. From Lemma 17, it follows that the number of bits required to write s_v for every vertex v is polynomial in the size of the input. Thus, the vector $\vec{\mathsf{s}}$ can be guessed in polynomial time.

Then, to verify the guess, it is sufficient to verify the **Bellman**, **lower-bound**, and **upper-bound** conditions for $\varphi_{\mathsf{FWMP}(\ell)}$. It is easy to see that the **Bellman** condition can be checked in polynomial time. Checking the **lower-bound** and **upper-bound** conditions, i.e., checking the almost-sure satisfaction of the threshold Boolean objective $\mathsf{FWMP}(\ell, \lambda)$ for appropriate thresholds λ in trap subgames in each $\vec{\mathsf{s}}$-class can be done in polynomial time [22].

Thus, the decision problem of $\mathbb{E}_v(\varphi_{\mathsf{FWMP}(\ell)}) \geq \lambda$ is in NP, and moreover, since there is exactly one value vector that satisfies the conditions in Theorem 7, the decision problem is, in fact, in UP. Analogously, the complement decision problem of $\mathbb{E}_v(\varphi_{\mathsf{FWMP}(\ell)}) < \lambda$ is also in UP. Hence, the expected $\varphi_{\mathsf{FWMP}(\ell)}$-value problem is in $\mathsf{UP} \cap \mathsf{coUP}$.

From the description of the optimal strategy in Lemma 9, it follows from Corollary 10 that the memory requirement for the expected $\varphi_{\mathsf{FWMP}(\ell)}$ objective is no greater than the memory requirement for the almost-sure satisfaction of the corresponding threshold objectives, which are ℓ and $|V| \cdot \ell$ for Player 1 and Player 2 respectively [22]. ◀

4.2 Expected bounded window mean-payoff value

We would like to apply the characterisation in Theorem 7 to φ_{BWMP} to show that the expected φ_{BWMP}-value problem is in $\mathsf{UP} \cap \mathsf{coUP}$, and thus, we show the existence of the bound $\mathsf{den}_{\mathsf{BWMP}}$ for the φ_{BWMP} objective. We show in Lemma 19 that the expected φ_{BWMP}-value $\mathfrak{s}_i$ of a class $\mathsf{S}(i)$ without boundary vertices is the mean payoff of a simple cycle in $\mathsf{S}(i)$.

▶ **Lemma 19.** *The expected φ_{BWMP}-value $\mathfrak{s}_i$ of vertices in a class $\mathsf{S}(i)$ without boundary vertices is equal to the mean-payoff value of a simple cycle in $\mathsf{S}(i)$. That is, $\mathfrak{s}_i$ is of the form $\frac{1}{j}(w(e_1) + \cdots + w(e_j))$ for some $j \leq |V|$ and edges $e_1, e_2, \ldots, e_j$ of a simple cycle.*

While Lemma 19 is analogous to Lemma 15 for $\varphi_{\mathsf{FWMP}(\ell)}$, the proof of Lemma 19 is more involved since the φ_{BWMP} objective requires one to consider windows of arbitrary lengths. In the proof, we make use of the fact that memoryless strategies suffice for Player 1 to play optimally for the almost-sure satisfaction of the BWMP objective [22]. In the resulting MDP (which has the same set of vertices as the game $\mathcal{G}_{\mathsf{S}(i)}$), we carefully analyse the resulting plays when Player 2 plays optimally. The following corollary of Lemma 19 states the bound $\underline{\mathsf{den}}_{\mathsf{BWMP}}$ for the φ_{BWMP} objective.

▶ **Corollary 20.** *The expected φ_{BWMP}-value of vertices in $\vec{\mathsf{s}}$-classes without boundary vertices can be written as $\frac{p}{q}$ where p and q are integers and $q \leq \widehat{w}^{|V|} \cdot |V|$.*

From Theorem 11, we get that the denominator of $\mathfrak{s}_i$ of each class $\mathsf{S}(i)$ in $\mathcal{G}$ is at most $2^{|V|} \cdot \widehat{\mathbb{P}}^{|V|^3} \cdot (\underline{\mathsf{den}}_{\mathsf{BWMP}})^{|V|}$, which is at most $2^{|V|} \cdot \widehat{\mathbb{P}}^{|V|^3} \cdot (\widehat{w}^{|V|} \cdot |V|)^{|V|}$.

▶ **Lemma 21.** *The expected φ_{BWMP}-value of each vertex in $\mathcal{G}$ can be written as $\frac{p}{q}$, where p, q are integers, and $q \leq 2^{|V|} \cdot \widehat{\mathbb{P}}^{|V|^3} \cdot (\widehat{w}^{|V|} \cdot |V|)^{|V|}$, and $-W \cdot q \leq p \leq W \cdot q$.*

We now state the main result of this section for the expected φ_{BWMP}-value problem.

▶ **Theorem 22.** *The expected φ_{BWMP}-value problem is in $\mathsf{UP} \cap \mathsf{coUP}$. Memoryless strategies suffice for Player 1. Player 2 requires infinite memory in general.*

Proof sketch. This proof follows a similar structure as the proof of Theorem 18. As before, the **Bellman** condition can be checked in polynomial time. Checking the **lower-bound** and **upper-bound** conditions involves checking almost-sure satisfaction of the Boolean objective BWMP for appropriate thresholds, which reduces to checking the satisfaction of BWMP in non-stochastic games [22], which in turn reduces to total supremum payoff [12], which is in $\mathsf{UP} \cap \mathsf{coUP}$ [27]. Both of these reductions are polynomial-time reduction, and hence, the expected φ_{BWMP}-value problem is in $\mathsf{UP} \cap \mathsf{coUP}$.

Memoryless strategies suffice for Player 1 for almost-sure satisfaction of $\mathsf{BWMP}(\lambda)$ [22]. Player 2 requires infinite memory in general for the $\mathsf{BWMP}(\lambda)$ objective even in non-stochastic games [12], which are a special case of stochastic games. Deterministic strategies suffice for both players. Hence, we get the memory requirements of an optimal strategy for the expected φ_{BWMP}-value problem using Corollary 10. ◀

5 Discussion

We discuss some concluding remarks about the relation of our work to previous work [16], which deals with the satisfaction of Boolean prefix-independent objectives. We also discuss practical implementations for window mean-payoff objectives and applicability of our technique to other prefix-independent objectives.

Comparison with [16]. In [16], it suffices to check the almost-sure satisfaction of the same Boolean objective ψ in all value classes. In contrast, for quantitative objectives, the threshold Boolean objective for which we check the almost-sure satisfaction depends on the guessed value of the value class ("Can Player 1 satisfy $\{\varphi > \mathfrak{r}_i - \varepsilon_\varphi\}$ with probability 1?"). Another key difference is that for Boolean objectives, the value classes without boundary vertices are precisely the extremal value classes, that is classes with values 0 and 1. In the case of quantitative objectives, there may be multiple intermediate value classes without boundary vertices, making reasoning about the correctness of the reduction more difficult.

We note that if we apply our approach to Boolean prefix-independent objectives (such as Büchi, coBüchi, parity) by viewing them as quantitative objectives mapping each play to 0 or 1, then we recover the algorithm given in [16].

Applicability to other prefix-independent objectives. Recall that in order to be able to apply our characterisation to a prefix-independent objective φ, we require bounds W_φ on the image of φ and den_φ on the denominators of the optimal expected φ-values of vertices in the game. These bounds can easily be derived individually for the following quantitative prefix-independent objectives, and thus, our technique applies to these objectives.

Mean-payoff objective. The mean-payoff value of any play is bounded between the minimum and maximum edge payoffs in the game, directly giving bounds on the image of the mean-payoff objective. Since deterministic memoryless optimal strategies exist for both players [24], the expected mean-payoff value is a solution of a stationary distribution in the Markov chain obtained by fixing memoryless strategies of both players. This gives denominator bounds for the expected mean-payoff values of vertices in the game.

Limsup and liminf objectives. Since the liminf objective is equivalent to $\mathsf{FWMP}(\ell)$ objective with window length $\ell = 1$ [21], and the limsup objective is the dual of the liminf objective, our analysis with window mean-payoff objectives generalises limsup and liminf objectives.

Positive frequency payoff objective [29]. Here, each vertex has a payoff, and this objective returns the maximum payoff among all vertices that are visited with positive frequency in an infinite play. This objective is prefix-independent, as the frequency of a vertex is independent of finite prefixes. The image of the objective is bounded between the minimum and maximum vertex payoffs. We observe that the value of vertices in a value class without boundary vertices is equal to the payoff of a vertex in the class, (giving a denominator bound for these vertices), and Theorem 11 uses this to give a denominator bound for vertices in value classes with boundary vertices.

Practical implementation. We discuss approaches to solve the expected φ-value problem for the window mean-payoff objectives in practice.

A trivial algorithm that works for both $\varphi_{\mathsf{FWMP}(\ell)}$ and φ_{BWMP} objectives is to iterate over all possible value vectors. For each value vector, we check if the conditions in Theorem 7 are satisfied, which can be done in polynomial time. Since there are exponentially many possible value vectors, this algorithm has an exponential running time in the worst-case.

Another technique is value iteration [15], which has been seen to be an anytime algorithm for the standard mean-payoff objective [34]. An anytime algorithm gives better precision the longer it is run, and can be interrupted any time. Given a game $\mathcal{G}$ with $|V|$ vertices, the expected $\varphi_{\mathsf{FWMP}(\ell)}$-value problem on $\mathcal{G}$ reduces to the expected liminf-value problem on a game $\mathcal{G}'$ with $|V|^\ell$ vertices, (that is, on an exponentially larger game graph). The liminf objective is a well-studied objective in the context of value iteration [11, 15]. We describe the reduction in [21], which also gives the expected $\varphi_{\mathsf{FWMP}(\ell)}$-values of vertices in $\mathcal{G}$.

Since the size of the graph $\mathcal{G}'$ is much bigger than that of $\mathcal{G}$, we would like to work with $\mathcal{G}'$ on-the-fly rather than explicitly constructing the entire graph. In [34], the authors show bounded value iteration for objectives such as reachability and mean-payoff. They also discuss that the algorithm can be extended to be asynchronous and use partial exploration. As future work, we would like to look at the practicality of on-demand asynchronous value iteration for the liminf objective, or even the window mean-payoff objectives $\varphi_{\mathsf{FWMP}(\ell)}$ and φ_{BWMP} directly. An interesting aspect of it would be to investigate heuristics and optimisations such as sound value iteration [37], optimistic value iteration [31], and topological value iteration [3] to speed up the practical running time.

References

1. D. Andersson and P. B. Miltersen. The Complexity of Solving Stochastic Games on Graphs. In *ISAAC*, volume 5878 of *Lecture Notes in Computer Science*, pages 112–121. Springer, 2009. doi:10.1007/978-3-642-10631-6_13.
2. K. R. Apt and E. Grädel. *Lectures in Game Theory for Computer Scientists*. Cambridge University Press, 2011.
3. M. Azeem, A. Evangelidis, J. Křetínský, A. Slivinskiy, and M. Weininger. Optimistic and Topological Value Iteration for Simple Stochastic Games. In *ATVA*, pages 285–302. Springer, 2022. doi:10.1007/978-3-031-19992-9_18.
4. C. Baier and J-P. Katoen. *Principles of model checking*. MIT Press, 2008.
5. B. Bordais, S. Guha, and J-F. Raskin. Expected Window Mean-Payoff. In *FSTTCS*, volume 150 of *LIPIcs*, pages 32:1–32:15, 2019. doi:10.4230/LIPIcs.FSTTCS.2019.32.
6. T. Brihaye, F. Delgrange, Y. Oualhadj, and M. Randour. Life is Random, Time is Not: Markov Decision Processes with Window Objectives. *Logical Methods in Computer Science*, Volume 16, Issue 4, December 2020. doi:10.23638/LMCS-16(4:13)2020.
7. V. Bruyère, E. Filiot, M. Randour, and J-F. Raskin. Meet your expectations with guarantees: Beyond worst-case synthesis in quantitative games. *Information and Computation*, 254:259–295, 2017. doi:10.1016/j.ic.2016.10.011.
8. T. Brázdil, V. Brožek, K. Chatterjee, V. Forejt, and A. Kučera. Markov Decision Processes with Multiple Long-run Average Objectives. *Logical Methods in Computer Science*, Volume 10, Issue 1, Feb 2014. doi:10.2168/LMCS-10(1:13)2014.
9. K. Chatterjee. Concurrent games with tail objectives. *Theoretical Computer Science*, 388(1):181–198, 2007. doi:10.1016/j.tcs.2007.07.047.
10. K. Chatterjee, L. Doyen, H. Gimbert, and T. A. Henzinger. Randomness for free. *Information and Computation*, 245:3–16, 2015. doi:10.1016/j.ic.2015.06.003.
11. K. Chatterjee, L. Doyen, and T. A. Henzinger. A Survey of Stochastic Games with Limsup and Liminf Objectives. In *ICALP*, pages 1–15. Springer, 2009. doi:10.1007/978-3-642-02930-1_1.
12. K. Chatterjee, L. Doyen, M. Randour, and J-F. Raskin. Looking at mean-payoff and total-payoff through windows. *Information and Computation*, 242:25–52, 2015. doi:10.1016/j.ic.2015.03.010.
13. K. Chatterjee and N. Fijalkow. A reduction from parity games to simple stochastic games. *EPTCS*, 54:74–86, 2011. doi:10.4204/eptcs.54.6.

14 K. Chatterjee and T. A. Henzinger. Reduction of stochastic parity to stochastic mean-payoff games. *Information Processing Letters*, 106(1):1–7, 2008. doi:10.1016/j.ipl.2007.08.035.

15 K. Chatterjee and T. A. Henzinger. Value Iteration. In *25 Years of Model Checking - History, Achievements, Perspectives*, LNCS 5000, pages 107–138. Springer, 2008. doi:10.1007/978-3-540-69850-0_7.

16 K. Chatterjee, T. A. Henzinger, and F. Horn. Stochastic Games with Finitary Objectives. In *MFCS*, pages 34–54. Springer, 2009. doi:10.1007/978-3-642-03816-7_4.

17 K. Chatterjee, Z. Křetínská, and J. Křetínský. Unifying Two Views on Multiple Mean-Payoff Objectives in Markov Decision Processes. *Logical Methods in Computer Science*, Volume 13, Issue 2, July 2017. doi:10.23638/LMCS-13(2:15)2017.

18 K. Chatterjee, T. Meggendorfer, R. Saona, and J. Svoboda. Faster Algorithm for Turn-based Stochastic Games with Bounded Treewidth. In *SODA*, pages 4590–4605, 2023. doi:10.1137/1.9781611977554.ch173.

19 A. Church. Application of Recursive Arithmetic to the Problem of Circuit Synthesis. *Journal of Symbolic Logic*, 28(4):289–290, 1963. doi:10.2307/2271310.

20 A. Condon. The complexity of stochastic games. *Information and Computation*, 96(2):203–224, 1992. doi:10.1016/0890-5401(92)90048-K.

21 L. Doyen, P. Gaba, and S. Guha. Expectation in Stochastic Games with Prefix-independent Objectives, 2024. doi:10.48550/arXiv.2405.18048.

22 L. Doyen, P. Gaba, and S. Guha. Stochastic Window Mean-payoff Games. In *FoSSaCS Part I*, volume 14574 of *LNCS*, pages 34–54. Springer, 2024. doi:10.1007/978-3-031-57228-9_3.

23 R. Durrett. *Probability: Theory and Examples*. Cambridge University Press, 2010.

24 A. Ehrenfeucht and J. Mycielski. Positional Strategies for Mean Payoff Games. *Int. Journal of Game Theory*, 8(2):109–113, 1979. doi:10.1007/BF01768705.

25 N. Fijalkow, N. Bertrand, P. Bouyer, R. Brenguier, A. Carayol, J. Fearnley, H. Gimbert, F. Horn, R. Ibsen-Jensen, N. Markey, B. Monmege, P. Novotny, M. Randour, O. Sankur, S. Schmitz, O. Serre, and M. Skomra. *Games on Graphs*. Online, 2024. doi:10.48550/arXiv.2305.10546.

26 P. Gaba and S. Guha. Optimising expectation with guarantees for window mean payoff in Markov decision processes. In *AAMAS*, pages 820–828. International Foundation for Autonomous Agents and Multiagent Systems / ACM, 2025. doi:10.5555/3709347.3743600.

27 T. M. Gawlitza and H. Seidl. Games through Nested Fixpoints. In *CAV*, pages 291–305. Springer, 2009. doi:10.1007/978-3-642-02658-4_24.

28 D. Gillette. *Stochastic games with zero stop probabilities*, pages 179–188. Princeton University Press, December 1958.

29 H. Gimbert and E. Kelmendi. Submixing and Shift-Invariant Stochastic Games. *International Journal of Game Theory*, 52(4):1179–1214, 2023. doi:10.1007/s00182-023-00860-5.

30 V. Gurvich and P. B. Miltersen. On the Computational Complexity of Solving Stochastic Mean-Payoff Games. *CoRR*, abs/0812.0486, 2008. arXiv:0812.0486.

31 A. Hartmanns and B. L. Kaminski. Optimistic Value Iteration. In *CAV*, pages 488–511. Springer, 2020. doi:10.1007/978-3-030-53291-8_26.

32 K. Hoffman and R. Kunze. *Linear Algebra*. Prentice-Hall, Inc., 1971.

33 J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Publishing Co., Inc., 1979.

34 J. Křetínský, T. Meggendorfer, and M. Weininger. Stopping Criteria for Value Iteration on Stochastic Games with Quantitative Objectives. In *LICS*, pages 1–14, 2023. doi:10.1109/LICS56636.2023.10175771.

35 D. A. Martin. The Determinacy of Blackwell Games. *The Journal of Symbolic Logic*, 63(4):1565–1581, 1998. doi:10.2307/2586667.

36 M. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley and Sons, 1994.

37 T. Quatmann and J-P. Katoen. Sound Value Iteration. In *CAV*, pages 643–661. Springer, 2018. doi:10.1007/978-3-319-96145-3_37.

38 L. S. Shapley. Stochastic Games. *Proceedings of the National Academy of Sciences*, 39(10):1095–1100, October 1953. doi:10.1073/pnas.39.10.1095.

39 U. Zwick and M. Paterson. The Complexity of Mean Payoff Games on Graphs. *Theoretical Computer Science*, 158(1&2):343–359, 1996. doi:10.1016/0304-3975(95)00188-3.

Just Verification of Mutual Exclusion Algorithms

Rob van Glabbeek
School of Informatics, University of Edinburgh, UK
School of Computer Science and Engineering, University of New South Wales, Sydney, Australia

Bas Luttik
Eindhoven University of Technology, The Netherlands

Myrthe S. C. Spronck[1]
Eindhoven University of Technology, The Netherlands

Abstract

We verify the correctness of a variety of mutual exclusion algorithms through model checking. We look at algorithms where communication is via shared read/write registers, where those registers can be atomic or non-atomic. For the verification of liveness properties, it is necessary to assume a completeness criterion to eliminate spurious counterexamples. We use justness as completeness criterion. Justness depends on a concurrency relation; we consider several such relations, modelling different assumptions on the working of the shared registers. We present executions demonstrating the violation of correctness properties by several algorithms, and in some cases suggest improvements.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Theory of computation → Concurrency; Theory of computation → Verification by model checking

Keywords and phrases Mutual exclusion, safe registers, regular registers, overlapping reads and writes, atomicity, safety, liveness, starvation freedom, justness, model checking, mCRL2

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2025.17

Related Version *Full Version*: `https://doi.org/10.48550/arXiv.2507.13198` [28]

Supplementary Material *Model (commit ff6122b)*: `https://github.com/mCRL2org/mCRL2/tree/master/examples/academic/non-atomic_registers`

Funding *Rob van Glabbeek*: Supported by Royal Society Wolfson Fellowship RSWF\R1\221008.

1 Introduction

The mutual exclusion problem is a fundamental problem in concurrent programming. Given $N \geq 2$ *threads*,[2] each of which may occasionally wish to access a *critical section*, a *mutual exclusion algorithm* seeks to ensure that at most one thread accesses its critical section at any given time. Ideally, this is done in such a way that whenever a thread wishes to access its critical section, it eventually succeeds in doing so. Many mutual exclusion algorithms have been proposed in the literature, and in general their correctness depends on assumptions one can make on the environment in which these algorithms will be running. The present paper aims to make these assumptions explicit, and to verify the correctness of some of the most popular mutual exclusion algorithms as a function of these assumptions.

Correctness properties of mutual exclusion algorithms. A thread that does not seek to execute its critical section is said to be executing its *non-critical section*. We regard *leaving the non-critical section* as getting the desire to enter the critical section. After this happens,

[1] Corresponding author
[2] What we call threads are in the literature frequently referred to as *processes* or *computers*. We use *threads* to distinguish between the real systems and our models of them, expressed in a process algebra.

© Rob van Glabbeek, Bas Luttik, and Myrthe S. C. Spronck;
licensed under Creative Commons License CC-BY 4.0

the thread is executing its *entry protocol*, the part of the mutual exclusion algorithm in which it negotiates with other threads who gets to enter the critical section first. The critical section occurs right after the entry protocol, and is followed by an *exit protocol*, after which the thread returns to its non-critical section. When in its non-critical section, a thread is not expected to communicate with the other threads in any way. Moreover, a thread may choose to remain in its non-critical section forever after. However, once a thread gains access to its critical section, it must leave it within a finite time, so as to make space for other threads.

The most crucial correctness property of a mutual exclusion algorithm is *mutual exclusion*: at any given time, at most one thread will be in its critical section. This is a safety property. In addition, a hierarchy of liveness properties have been considered. The weakest one is *deadlock freedom*: Whenever at least one thread is running its entry protocol, eventually some thread will enter its critical section. This need not be one of the threads that was observed to be in its entry protocol. A stronger property is *starvation freedom*: whenever a thread leaves its non-critical section, it will eventually enter its critical section. A yet stronger property, called *bounded bypass*, augments starvation freedom with a bound on the number of times other threads can gain access to the critical section before any given thread in its entry protocol.

In this paper we check for over a dozen mutual exclusion protocols, and for six possible assumptions on the environment in which they are running, whether they satisfy mutual exclusion, deadlock freedom and starvation freedom. We will not investigate bounded bypass, nor other desirable properties of mutual exclusion protocols, such as *first-come-first-served*, *shutdown safety*, *abortion safety*, *fail safety* and *self-stabilisation* [38].

Memory models.[3] In the mutual exclusion algorithms considered here, the threads communicate with each other solely by reading from and writing to shared registers. The main assumptions on the environment in which mutual exclusion algorithms will be running concern these registers. It is frequently assumed that (read and write) operations on registers are "undividable", meaning that they cannot overlap or interleave each other: if two threads attempt to perform an operation on the same register at the same time, one operation will be performed before the other. This assumption, sometimes referred to as *atomicity*, is explicitly made in Dijkstra's first paper on mutual exclusion [20]. Atomicity is sometimes conceptualised as operations occurring at a single moment in time. We instead acknowledge that operations have duration. Consequently, if operations cannot overlap in time, then, when multiple operations are attempted simultaneously, the one performed first must postpone the occurrence of the others by at least its own duration. One operation postponing another is called *blocking* [18].

Deviating from Dijkstra's original presentation, several authors have considered a variation of the mutual exclusion problem where the atomicity assumption is dropped [36, 47, 37, 38, 52, 53, 4, 6]. Attempted operations can then occur immediately, without blocking each other. We say these operations are *non-blocking*. In this context, read and write operations may be *concurrent*, i.e. overlap in time. We must then consider the consequences of operations overlapping each other.

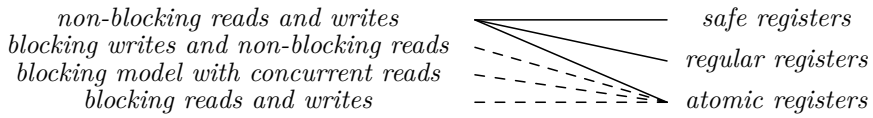
[3] A *memory model* describes the interactions of threads through memory and their shared use of the data. The models reviewed here differ in the degree in which different register accesses exclude each other, and in what values a register may return in case of overlapping reads and writes. In this paper, we do not consider *weak memory models*, that allow for compiler optimisations, and for reads to sometimes fetch values that were already changed by another thread. In [8] is has been shown that mutual exclusion cannot be realised in weak memory models, unless those models come with *memory fences* or *barriers* that can be used to undermine their weak nature.

In [39, 40], Lamport proposes a hierarchy of three memory models in this context, specifically for single-writer multi-reader (SWMR) registers; such registers are owned by one thread, and only that thread is capable of writing to it. Crucial for these definitions is the assumption that every register has a domain, and a read of that register always yields a value from that domain. It is also important to note that threads can only perform a single operation at a time, meaning that a thread's operations can never overlap each other.

- A **safe** register guarantees merely that a read that is not concurrent with any write returns the most recently written value.
- A **regular** register guarantees that any read returns either the last value written before it started, or the value of any overlapping write, if there is one.
- An **atomic** register guarantees that reads and writes behave as though they occur in some total order. This total order must comply with the real-time ordering of the operations: if operation a ends before operation b begins, then a must be ordered before b.

These three memory models form a hierarchy, in the sense that any atomic register is regular, and any regular one is safe. When we merely know that a register is safe, a read that overlaps with any write might return any value in the domain of the register. In Section 3 we discuss the generalisation of these memory models to multi-writer multi-reader (MWMR) registers, ones that can be written and read by all threads.

Besides blocking and non-blocking registers, as explained above, we consider two intermediate memory models. The *blocking model with concurrent reads* requires (1) any scheduled read or write to await the completion of any write that is in progress, and (2) any scheduled write to await the completion of any unfinished read. However, reads from different threads need not wait for each other and may overlap in time without ill effects. In the model of *non-blocking reads*,[4] we have (1) but not (2). This model, where writes block reads but reads do not block writes, may apply when writes can abort in-progress reads, superseding them.



In this paper, we model six different memory models, which are illustrated above. The blocking aspect of our memory models is captured via different concurrency relations (Section 5). The distinction between safe, regular and atomic registers is captured via three different process algebraic models (Section 3). Since the safe/regular/atomic distinction is only relevant in models that allow writes to overlap reads and writes, we only make it for the non-blocking model; for the other three memory models we reuse our atomic register models.

Completeness criteria. In previous work [50], we checked the mutual exclusion property of several algorithms, with safe, regular and atomic MWMR registers, through model checking with the mCRL2 toolset [16]. We did not check the liveness properties at that time; the presence of certain infinite loops in our models introduced spurious counterexamples to such properties, which hindered our verification efforts. As an example of what we call a "spurious counterexample", we frequently found violations to starvation freedom where one thread, i, never obtained access to its critical section because a different thread, j, was endlessly repeating a busy wait, or some other infinite cycle which should reasonably not prevent

[4] In this terminology, from [18], a *blocking read* blocks a write; it does not refer to a read that is blocked.

i from progressing to its critical section. Yet, the model checker does not know this, and can therefore only conclude that the property is not satisfied. In this paper, we extend our previous work by addressing this problem and checking liveness properties as well.

One method for discarding spurious counterexamples from verification results is applying completeness criteria: rules for determining which paths in the model represent real executions of the modelled system. By ensuring that all spurious paths are classified as incomplete and only taking complete paths into consideration when verifying liveness properties, we can circumvent the spurious counterexamples. Of course, one must take care not to discard true system executions by classifying those as incomplete. The completeness criterion must therefore be chosen with care. Examples of well-known completeness criteria are weak fairness and strong fairness. Weak fairness assumes that every task[5] that eventually is perpetually enabled must occur infinitely often; strong fairness assumes that if a task is infinitely often enabled it must occur infinitely often [42, 5, 27]. In effect, making a fairness assumptions amounts to assuming that if something is tried often enough, it will always eventually succeed [27]. In that sense, these assumptions, even weak fairness, are rather strong, and may well result in true system executions being classified as incomplete. In this paper, we therefore use the weaker completeness criterion *justness* [27, 24, 14].

Unlike weak and strong fairness, justness takes into account how different actions in the model relate to each other. Informally, it says that if an action a can occur, then eventually a occurs itself, or a different action occurs that interferes with the occurrence of a. The underlying idea of justness is that the different components that make up a system must all be capable of making progress: if thread i wants to perform an action entirely independent of the actions performed by j, then there can be no interference. However, if both threads are interacting with a shared register, then we may decide that one thread writing to the register can prevent the other from reading it at the same time, or vice versa. Which actions interfere with each other is a modelling decision, dependent on our understanding of the real underlying system. It is formalised through a *concurrency relation*, which must adhere to some restrictions. In this paper we propose four concurrency relations, each modelling one of the four major memory models reviewed above: non-blocking reads and writes, blocking writes and non-blocking reads, the blocking model with concurrent reads, and blocking reads and writes.

Model checking. Traditionally, mutual exclusion algorithms have been verified by pen-and-paper proofs using behavioural reasoning. As remarked by Lamport [38], "the behavioral reasoning used in our correctness proofs, and in most other published correctness proofs of concurrent algorithms, is inherently unreliable". This is especially the case when dealing with the intricacies of non-atomic registers.[6] This problem can be alleviated by automated formal verification; here we employ model checking.

While the precise modelling of the algorithms, the registers and the employed completeness criterion requires great care, the subsequent verification requires a mere button-push and some patience. Since our model checker traverses the entire state-space of a protocol, the verified protocols and all their registers need to be finite. This prevented us from checking

[5] What constitutes a *task* differs from paper to paper; hence there are multiple flavours of strong and weak fairness; here a task could be a read or write action of a certain thread on a certain register.

[6] A good illustration of unreliable behavioural reasoning is given in [25, Section 21], through a short but fallacious argument that the mutual exclusion property of Peterson's mutual exclusion protocol, which is known to hold for atomic registers, would also hold for safe registers. We challenge the reader to find the fallacy in this argument before looking at the solution.

the bakery algorithm [36], as it is one of the few mutual exclusion protocols that employs an unbounded state space. Moreover, those algorithms that work for N threads, for any $N \in \mathbb{N}$, could be checked for small values of N only; in this paper we take $N = 3$. Consequently, any failure of a correctness property that shows up only for > 3 threads will not be caught here.

As stated, we employed these methods in previous work to check mutual exclusion algorithms. Although there we checked only safety properties, and did not consider the blocking aspects of memory, this already gave interesting results. For instance, we showed that Szymanski's flag algorithm from [52], even when adapted to use Booleans, violates mutual exclusion with non-atomic registers. Here, we expand this previous work by checking deadlock freedom and starvation freedom in addition to mutual exclusion, and by including blocking into our memory models. In total, we check the three correctness properties of over a dozen mutual exclusion algorithms, for six different memory models. Among others, we cover Aravind's BLRU algorithm [6], Dekker's algorithm [19, 3] and its RW-safe variant [15], and Szymanski's 3-bit linear wait algorithm [53]. In some cases where we find property violations, we suggest fixes to the algorithms so that the properties are satisfied.

2 Preliminaries

A *labelled transition system* (LTS) is a tuple $(\mathcal{S}, \mathit{Act}, \mathit{init}, \mathit{Trans})$ in which $\mathcal{S}$ is a finite set of states, Act is a finite set of actions, $\mathit{init} \in \mathcal{S}$ is the initial state, and $\mathit{Trans} \subseteq \mathcal{S} \times \mathit{Act} \times \mathcal{S}$ is a transition relation. We write $s \xrightarrow{a} s'$ for $(s, a, s') \in \mathit{Trans}$. We say an action a is *enabled* in a state s if there exists a state s' such that $(s, a, s') \in \mathit{Trans}$.

A *path* π is a non-empty, potentially infinite alternating sequence of states and actions $s_0 a_1 s_1 a_2 \ldots$, with $s_0, s_1, \ldots \in \mathcal{S}$ and $a_1, a_2, \ldots \in \mathit{Act}$, such that if π is finite, then its last element is a state, and for all $i \in \mathbb{N}$, $s_i \xrightarrow{a_{i+1}} s_{i+1}$. The first state of π is its *initial state*. The *length* of π is the number of transitions in it.

We use a notion of parallel composition that is taken from Hoare's CSP [33], where synchronisation between components is enforced on all shared actions. It is defined as follows: For some $k \geq 1$, let $P_1, \ldots, P_k$ be LTSs, where $P_i = (\mathcal{S}_i, \mathit{Act}_i, \mathit{init}_i, \mathit{Trans}_i)$ for all $1 \leq i \leq k$. The *parallel composition* $P_1 \parallel \ldots \parallel P_k$ of $P_1, \ldots, P_k$ is the LTS $P = (\mathcal{S}, \mathit{Act}, \mathit{init}, \mathit{Trans})$ in which $\mathcal{S} = \mathcal{S}_1 \times \ldots \times \mathcal{S}_k$, $\mathit{Act} = \bigcup_{1 \leq i \leq k} \mathit{Act}_i$, $\mathit{init} = (\mathit{init}_1, \ldots, \mathit{init}_k)$, and a transition $((s_1, \ldots, s_k), a, (s'_1, \ldots, s'_k))$ is in Trans if, and only if, $a \in \mathit{Act}$ and the following are true for all $1 \leq i \leq k$: if $a \notin \mathit{Act}_i$, then $s_i = s'_i$, and if $a \in \mathit{Act}_i$, then $(s_i, a, s'_i) \in \mathit{Trans}_i$.
Note that by this definition of Trans, if an action is in the action set of a component but not enabled by that component in a particular state of the parallel composition, then the composition cannot perform a transition labelled with that action.

As mentioned in the introduction, the completeness criterion we use for our liveness verification is a variant of *justness* [24, 27]. Specifically, while justness is originally defined on transitions, we here define it on action labels, an adaption we take from [14]. As stated earlier, the definition of justness relies on the notion of a concurrency relation.

▶ **Definition 1.** Given an LTS $(\mathcal{S}, \mathit{Act}, \mathit{init}, \mathit{Trans})$, a relation $\smile\!\!\bullet \subseteq \mathit{Act} \times \mathit{Act}$ is a *concurrency relation* if, and only if:

- $\smile\!\!\bullet$ is irreflexive.
- For all $a \in \mathit{Act}$, if π is a path from a state $s \in \mathcal{S}$ to a state $s' \in \mathcal{S}$ such that a is enabled in s and $a \smile\!\!\bullet b$ for all $b \in \mathit{Act}$ occurring on π, then a is enabled in s'.

A concurrency relation may be asymmetric. We often reason about the complement of $\smile\!\!\bullet$, $\not\smile\!\!\bullet$. Read $a \smile\!\!\bullet b$ as "a is independent from b" and $a \not\smile\!\!\bullet b$ as "b interferes with/postpones a".

▶ **Observation 2.** Concurrency relations can be refined by removing pairs; a subset of a concurrency relation is still a concurrency relation.

Informally, justness says that a path is complete if whenever an action a is enabled along the path, there is eventually an occurrence of an action (possibly a itself) that interferes with it. This can be weakened by defining a set of *blockable actions*, for which this restriction does not hold; a blockable action may be enabled on a complete path without there being a subsequent occurrence of an interfering action. In this paper, the action of a thread to leave its non-critical section will be blockable. This way we model that a thread may choose to never take that option. We give the formal definition of justness, incorporating the blockable actions. We represent the set of blockable actions as $\mathcal{B}$. Its complement, $\overline{\mathcal{B}}$, is defined as $Act \setminus \mathcal{B}$, given a set of actions Act.

▶ **Definition 3.** A path π in an LTS $(\mathcal{S}, Act, init, Trans)$ satisfies $\mathcal{B}$-$\smile\!\!\!\bullet$-*justness of actions* ($JA^{\smile\!\!\!\bullet}_{\mathcal{B}}$) if, and only if, for each suffix π' of π, if an action $a \in \overline{\mathcal{B}}$ is enabled in the initial state of π', then an action $b \in Act$ occurs in π' such that $a \not\smile\!\!\!\bullet\, b$.

We say that a property is satisfied on a model under $JA^{\smile\!\!\!\bullet}_{\mathcal{B}}$ if it is satisfied on every path of that model, starting from the model's initial state, that satisfies $JA^{\smile\!\!\!\bullet}_{\mathcal{B}}$. If $\mathcal{B}$ and $\smile\!\!\!\bullet$ are clear from the context, we simply say that a path that satisfies $JA^{\smile\!\!\!\bullet}_{\mathcal{B}}$ is *just*.

3 Register models

In [50], we presented process-algebraic models of MWMR safe, regular and atomic registers. Through the semantics of the process algebra, this determines an LTS for each register of a given kind. In this paper, we use the same definitions for the three register types, but we have altered the process-algebraic models to be more compact and better facilitate the definition of the concurrency relations. The process-algebraic models can be found in Appendix A; here we merely summarise the key design decisions.

A register model represents a multi-reader multi-writer read-write register that allows every thread to read from and write to it. However, every thread may only perform a single operation on the register at a time. The register model specifies the behaviour of the register in response to operations performed by threads. Here we presuppose two disjoint finite sets: $\mathbb{T}$ of thread identifiers (thread id's) and $\mathbb{R}$ of register identifiers (register id's). Additionally, for every $r \in \mathbb{R}$ we reference the set $\mathbb{D}_r$ of all data values that the register r can hold.

Recall that read and write operations take time, and may hence be concurrent. Therefore we represent a single operation with two actions: an *invocation* to indicate the start of the operation, and a *response* to indicate its end. Two operations are concurrent if the interval between their respective invocations and responses overlaps. The interface of a register is represented by the following actions: a read by thread $t \in \mathbb{T}$ of register $r \in \mathbb{R}$ that returns value $d \in \mathbb{D}_r$ is a start read action $sr_{t,r}$ followed by a finish read action $fr_{t,r}(d)$; a write of value $d \in \mathbb{D}_r$ by a thread $t \in \mathbb{T}$ to register $r \in \mathbb{R}$ is a start write action $sw_{t,r}(d)$ followed by a finish write action $fw_{t,r}$. In addition to this interface, which the threads can use to perform operations on the register, the regular and atomic models also use *register local actions*; these are internal actions by the register that are used to model the correct behaviour.

A register model requires some finite amount of memory to store a representation of relevant past events. We store this in what we call the *status object*, which features a finite set $\mathbb{S}$ of possible states. We abstract away from the exact implementation; for this presentation, all that is relevant is which information can be retrieved from it. Amongst others, we use the following *access functions*, which are local to any given register r:

- $stor : \mathbb{S} \to \mathbb{D}_r$, the value that is currently stored in the register.
- $wrts : \mathbb{S} \to 2^{\mathbb{T}}$, the set of thread id's of threads that have invoked a write operation on this register that has not yet had its response.

Any occurrence of a register action a induces a state change $s \xrightarrow{a} s'$, resulting in an *update* to these access functions. For instance, the actions $sw_{t,r}(d)$ and $fw_{t,r}$ cause the updates $wrts(s') = wrts(s) \cup \{t\}$ and $wrts(s') = wrts(s) \setminus \{t\}$, respectively.

3.1 Safe MWMR registers

To extend the single-writer definition of safe registers to a multi-writer one, we follow Lamport in assuming that a concurrent read cannot affect the behaviour of a read or write. Lamport's SWMR definitions consider how concurrent writes affect reads, but not how concurrent writes affect writes. Here we follow Raynal's approach for safe registers: a write that is concurrent with another write sets the value of the register to some arbitrary value in the domain of the register [48]. We can summarise the behaviour of MWMR safe registers with four rules:

1. A read not concurrent with any writes on the same register returns the value most recently written into the register.
2. A read concurrent with one or more writes on the same register returns an arbitrary value in the domain of the register.
3. A write not concurrent with any other write on the same register results in the intended value being set.
4. A write concurrent with one or more other writes on the same register results in an arbitrary value in the domain of the register being set.

In our model of a safe register r, its status object maintains a Boolean variable *ovrl* for each thread id, telling whether an ongoing read or write action of this thread overlapped with a write by another thread. The value of *ovrl* is updated in a straightforward way each time r experiences a register interface action $sr_{t,r}$, $fr_{t,r}(d)$, $sw_{t,r}(d)$ or $fw_{t,r}$, aided by the access function *wrts*. Using this function, our model can determine which of the above four rules applies when a read or write finishes, and behave accordingly.

3.2 Regular MWMR registers

We wish to define regular MWMR registers as an extension of Lamport's definition of SWMR regular registers: a read returns either the last written value before the read began, or the value of any concurrent write, if there is one. This is non-trivial; in [50] we present one extension and compare it to four different suggestions from [49]. The complexity comes from determining what the last written value is, given that writes may be concurrent with each other. Here, following [50], we require that all threads see the same global ordering on writes once those writes have completed. Hence, if two writes w_1 and w_2 occur concurrently, and after their completion, but before the invocation of any other write, there are two reads r_1 and r_2, then either both r_1 and r_2 see w_1's value as the last written value, or they both see w_2's value as the last written value. We generate the global ordering through the register local order write action $ow_{t,r}$, which is scheduled between the start write action $sw_{t,r}(d)$ and the finish write $fw_{t,r}$. This action does not represent any true internal behaviour by the register; the interleaving of order write actions from various threads merely determines the global ordering. Given a read, we say the "last written" value this read sees, and hence the value this read may return in addition to those of overlapping writes, is the intended value of the write whose ordering was the most recent before the read's invocation.

In our process algebraic model, the value *stor* of the register is set when an action $ow_{t,r}$ occurs. During any read action of a thread t, that is, between $sr_{t,r}$ and $fr_{t,r}(d)$, the register model builds a set of the possible return values on the fly. When the read starts, this set is initialised to *stor* and the intended value of every active write. Subsequently, whenever a write occurs, its intended value is added to the set. This way, at the finish read, the set will contain exactly those values that the read could return.

3.3 Atomic MWMR registers

Lamport's definition of SWMR atomic registers, namely that the register must behave as though reads and writes occur in some strict order, is directly applicable to the MWMR case. We reuse the register local order write action from the regular register model, and add the similar order read action $or_{t,r}$ for read operations. This way, we generate an ordering on all operations. In our process-algebraic model, *stor* is updated when the $ow_{t,r}$ occurs, similar to regular registers. For read operations, the value of *stor* when $or_{t,r}$ occurs is remembered, and returned at the matching response.

4 Thread-register models

In our models, we combine processes representing both threads and registers. Similar to how register processes may contain register local actions, threads may contain *thread local actions*. Crucially, the local actions of both registers and threads are not involved in any communication, meaning that the only way for two threads to communicate is by writing to and reading from registers using the register interface actions.

The register models are mostly independent of the algorithm that we analyse with them; the algorithm merely dictates a register's identifier, domain, and initial value. However, the thread models are fully dependent on the modelled algorithm, which dictates their behaviour. Therefore, we cannot present an algorithm-independent thread process. Instead, we presuppose the existence of an LTS $T_t = (\mathcal{S}_t, Act_t, init_t, Trans_t)$ and a set of thread local actions $TLoc_t$ for every $t \in \mathbb{T}$ such that $Act_t = \{sr_{t,r}, fr_{t,r}(d), sw_{t,r}(d), fw_{t,r} \mid r \in \mathbb{R}, d \in \mathbb{D}_r\} \cup TLoc_t$. We assume that all sets of thread local actions are pairwise disjoint and that all thread LTSs T_t satisfy two properties, representing the reasonable implementation of read and write operations. Firstly, on all paths from $init_t$, each transition labelled $sr_{t,r}$ for some $r \in \mathbb{R}$ must go to a state where exactly the actions $fr_{t,r}(d)$ for all $d \in \mathbb{D}_r$ are enabled. Similarly, all transitions labelled $sw_{t,r}(d)$ for some $r \in \mathbb{R}, d \in \mathbb{D}_r$ must go to states where only $fw_{t,r}$ is enabled. Secondly, transitions labelled $fr_{t,r}(d)$ or $fw_{t,r}$ are only enabled in these states.

We combine thread and register LTSs into *thread-register models*. For the following definition, we let $R_r = (\mathcal{S}_r, Act_r, init_r, Trans_r)$ be the LTS associated with each $r \in \mathbb{R}$.

▶ **Definition 4.** A *thread-register model* is a six-tuple $(\mathcal{S}, Act, init, Trans, thr, reg)$, such that $(\mathcal{S}, Act, init, Trans)$ is a parallel composition of thread and register LTSs and

- $thr\colon Act \to \mathbb{T}$ is a mapping from actions to thread id's; and
- $reg\colon Act \to \mathbb{R} \cup \{\bot\}$ is a mapping from actions to register id's and the special value $\bot \notin \mathbb{R}$.

thr and *reg* are defined in the obvious way, e.g., $thr(sr_{t,r}) = t$ and $reg(sr_{t,r}) = r$. Crucially, for a thread local action a, $reg(a) = \bot$.

Note that by our construction of the thread and register LTSs, every action in Act appears in at most two components of the parallel composition. Specifically, for all $t \in \mathbb{T}$ and $a_t \in TLoc_t$, a_t appears only in T_t; for all $t \in \mathbb{T}, r \in \mathbb{R}$, $or_{t,r}$ and $ow_{t,r}$ appear only in R_r; and for all $t \in \mathbb{T}, r \in \mathbb{R}$ and $d \in \mathbb{D}_r$, $sr_{t,r}, fr_{t,r}(d), sw_{t,r}(d)$ and $fw_{t,r}$ appear exactly in T_t and R_r.

5 Justness for thread-register models

In order to obtain a suitable notion of justness for our thread-register models, we need to choose both $\mathcal{B}$ and $\smile\!\!\bullet$. Only thread local actions will be blockable; we define $\mathcal{B}$ in Section 6.

The concurrency relation, on the other hand, should relate the register interface actions. This is how we represent whether it is reasonable for one thread's operations on a register to interfere with (and thereby postpone) another thread's operations on that register. We use four different concurrency relations in our verifications, representing the four different models of blocking described in Section 1. These concurrency relations do not reference the thread local actions outside of the *thr* mapping, so we can already present these relations before giving more details on the precise models. To establish that the relations we present are indeed concurrency relations, we first establish a property of our models. We call this property *thread consistency*.

▶ **Definition 5.** An LTS $(\mathcal{S}, Act, init, Trans)$ is *thread consistent* with respect to a mapping $thr : Act \to \mathbb{T}$ if, and only if, for all states $s \in \mathcal{S}$, if an action $a \in Act$ is enabled in s and there exists a transition $s \xrightarrow{b} s'$ for some $s' \in \mathcal{S}, b \in Act$ such that $thr(a) \neq thr(b)$, then a is also enabled in s'.

The correctness of our concurrency relations (cf. Definition 1) relies on our thread-register models being thread-consistent. The proof of this fact is given in Appendix B.

▶ **Lemma 6.** Let $M = (\mathcal{S}, Act, init, Trans, thr, reg)$ be a thread-register model. Then the LTS $(\mathcal{S}, Act, init, Trans)$ is thread consistent with respect to the mapping *thr*.

For the definitions of our four concurrency relations, we fix a thread-register model $M = (\mathcal{S}, Act, init, Trans, thr, reg)$. We also introduce two predicates on Act: $sr?$ and $sw?$. For an action $a \in Act$, these are defined as:

$$sr?(a) = \exists_{t\in\mathbb{T}, r\in\mathbb{R}}.(a = sr_{t,r}) \qquad sw?(a) = \exists_{t\in\mathbb{T}, t\in\mathbb{R}, d\in\mathbb{D}_r}.(a = sw_{t,r}(d)).$$

The *thread interference relation*, $\smile\!\!\bullet_T$, expresses that every action is independent from every other action *unless* the two actions belong to the same thread; every two actions by the same thread interfere with each other. It captures the memory model with non-blocking reads and writes. This is the coarsest concurrency relation we will use.

▶ **Definition 7.** $\smile\!\!\bullet_T = \{(a, b) \mid a, b \in Act, thr(a) \neq thr(b)\}$

▶ **Lemma 8.** $\smile\!\!\bullet_T$ is a concurrency relation for M.

This follows by a straightforward application of Lemma 6. The details are in [28, Appendix D].

The model with blocking writes and non-blocking reads is captured by the *signalling reads relation*, $\smile\!\!\bullet_S$.

▶ **Definition 9.** $\smile\!\!\bullet_S = \smile\!\!\bullet_T \setminus \{(a, b) \mid a, b \in Act, sr?(a) \vee sw?(a), sw?(b), reg(a) = reg(b)\}$

Intuitively, this is the same as $\smile\!\!\bullet_T$ except that one thread starting a write to a register can interfere with a write to or a read from that same register by another thread. However, a read cannot interfere with another thread's read or write. This concurrency relation has a precedent in [21, 14]. There, reads are modelled as *signals*, which differ from standard actions in that they do not block any other actions. Hence the name of this relation.

The blocking model with concurrent reads is captured by the *interfering reads relation*, $\smile\!\!\bullet_I$. This is a further refinement from $\smile\!\!\bullet_S$, where a start read can interfere with a start write on the same register, but cannot interfere with a start read.

▶ **Definition 10.** $\smile\!\!\bullet_I = \smile\!\!\bullet_S \setminus \{(a, b) \mid a, b \in Act, sw?(a), sr?(b), reg(a) = reg(b)\}$

This goes with the idea that performing a write on a memory location can only be done when the memory is reserved: repeated reads can prevent the memory from being reserved for a write, but as long as there is no write all the reads can take place concurrently.

Finally, the model of blocking reads and writes is captured by the *all interfering relation*, $\smile\!\!\bullet_A$, a refinement of $\smile\!\!\bullet_I$ where a start read can also interfere with another start read.

▶ **Definition 11.** $\smile\!\!\bullet_A = \smile\!\!\bullet_I \setminus \{(a,b) \mid a,b \in Act, sr?(a), sr?(b), reg(a) = reg(b)\}$

In this model, every operation on a register fully reserves that register for only that operation, and hence can prevent any other operation from taking place at the same time.

▶ **Lemma 12.** $\smile\!\!\bullet_S$, $\smile\!\!\bullet_I$ and $\smile\!\!\bullet_A$ are concurrency relations for M.

Proof. This follows from Lemma 8 and Observation 2. ◀

As stated in the introduction, we capture six memory models. We obtain three variants of the non-blocking model by combining the $\smile\!\!\bullet_T$ relation with the safe, regular and atomic register models. The remaining three memory models are represented by combining $\smile\!\!\bullet_S$, $\smile\!\!\bullet_I$ and $\smile\!\!\bullet_A$ with the atomic register model. In [28, Appendix E], we formally characterise just paths in our thread-register models for all six variants. In [28, Appendix F], we prove that using the atomic register model, which allows overlapping writes, for the three memory models with blocking writes is sound for verification purposes.

6 Verification

Below, we collect over a dozen mutual exclusion algorithms from the literature. We also present altered versions of several algorithms, incorporating fixes we propose. All these algorithms, and the registers themselves, have been translated to the process algebra mCRL2 [30]. The mCRL2 files are available as supplementary material. We give the most important design decisions regarding this translation here; further details can be found in [28, Appendix G]. The only operations on registers we allow are reading and writing. Hence, more complicated statements that may have been present in the original presentation of an algorithm, such as compare-and-swap instructions, are converted into these primitive operations. A statement like "**await** $\forall_{i \in \mathbb{T}} : x(i)$", where x is some condition on a thread id i, is modelled as a recursive process that checks each thread id from smallest to largest, waiting for each until $x(i)$ is satisfied. Where an algorithm does not specify the initial value of a register, we take the lowest value from the given domain.

We use c_t and nc_t, with $t \in \mathbb{T}$, as thread local actions. These actions represent a thread entering its critical section and leaving its non-critical section, respectively. We define $\mathcal{B} = \{nc_t \mid t \in \mathbb{T}\}$ to capture that a thread may always choose to remain in its non-critical section indefinitely; this is an important assumption underlying the correctness of mutual exclusion protocols.

We did our verification with the mCRL2 toolset [16]. To this end, we encoded mutual exclusion, deadlock freedom, and starvation freedom in the modal μ-calculus, the logic used to represent properties in the mCRL2 toolset. We used the patterns from [51] to incorporate the justness assumption into our formulae for deadlock freedom and starvation freedom. The full modal μ-calculus formulae appear in [28, Appendix H] (and also as supplementary material). Besides the correctness properties discussed in Section 1, Dijkstra [20] requires that the correctness of the algorithms may not depend on the relative speeds of the threads. This requirement is automatically satisfied in our approach, since we allow all possible interleavings of thread actions in our models.

We checked mutual exclusion, deadlock freedom, and starvation freedom. If mutual exclusion is not satisfied, we do not care about the other two properties. Additionally, if deadlock freedom is not satisfied, we know that starvation freedom is not satisfied either. We can therefore summarise our results in a single table: X if none of the three properties are satisfied, M if only mutual exclusion is satisfied, D if only mutual exclusion and deadlock freedom hold, and S if all three are satisfied. See Table 1. As stated previously, we verify liveness properties under justness, where we employ $\smile\!\!\bullet_T$ for safe and regular registers and all four concurrency relations $\smile\!\!\bullet_T$, $\smile\!\!\bullet_S$, $\smile\!\!\bullet_I$ and $\smile\!\!\bullet_A$ for atomic registers. We checked with 2 threads for algorithms designed for 2 threads, and with 3 for all others. We restrict ourselves to at most 3 threads because, due to the state-space explosion problem, even models with only 3 threads frequently take hours or even days to check these properties on.

We list the origin of each algorithm in the table; the results of verifying our proposed alternate versions are indicated by "alt.". For the algorithms we discuss in detail, we include their pseudocode. Therein we merely present the entry and exit protocols of an algorithm, separated by the instruction **critical section**. Implicitly these instructions alternate with the non-critical section, and may be repeated indefinitely. We use N for the number of threads. As identifiers for threads, we use the integers $0 \ldots N{-}1$. So $\mathbb{T} = \{0, \ldots, N{-}1\}$. When presenting pseudocode, we give the algorithm for an arbitrary thread i. When $N = 2$, we use the shorthand notation $j = 1 - i$.

In the subsequent sections, we discuss the most interesting results. Pseudocode and further discussion of the algorithms not covered here appear in [28, Appendix I].

6.1 Impossibility of liveness with $\smile\!\!\bullet_I$

Perhaps the most notable result in Table 1 is that no algorithm satisfies either liveness property under $JA_{\mathcal{B}}^{\smile\!\!\bullet_I}$ or $JA_{\mathcal{B}}^{\smile\!\!\bullet_A}$. Since $\smile\!\!\bullet_A$ is a refinement of $\smile\!\!\bullet_I$, we focus on the behaviour for $\smile\!\!\bullet_I$. When we take $\smile\!\!\bullet_I$ as our concurrency relation, then one thread's read of a register can interfere with another thread's write to that same register. It turns out that when this is the case, starvation freedom is impossible for algorithms that rely on communication via registers. The following argument is adapted from [23, 25]. Assume that *Alg* is an algorithm that satisfies starvation freedom. Let i and j be different threads, and assume that all other threads, if any, stay in their non-critical section forever. Since *Alg* is starvation-free, thread i must be capable of freely entering the critical section if thread j is not competing for access. Hence, thread j must communicate its interest in the critical section to thread i as part of its entry protocol. Since reading from and writing to registers is the only form of communication we allow, thread j must, in its entry protocol, write to some register *reg*, which i must read in its own entry protocol. As long as i does not read j's interest from *reg*, thread i can enter the critical section freely. Therefore, if thread i's read of *reg* can block thread j's write to *reg*, thread i can infinitely often access the critical section without ever letting thread j communicate its interest, thus never letting thread j enter.

For this argument it is crucial that right after i's read of *reg*, thread i enters and then leaves the critical section and returns to its entry protocol, where it engages in another read of *reg*, so quickly that thread j has not yet started its write to *reg* in the meantime. This uses the requirement on mutual exclusion protocols that their correctness may not depend on the relative speeds of the threads. Without that requirement one can easily achieve starvation freedom even with blocking reads, as demonstrated in [25].

The argument above explains why starvation freedom is never satisfied for $\smile\!\!\bullet_I$ or $\smile\!\!\bullet_A$. However, it does not explain why we also never observe deadlock freedom. After all, in the execution sketched above, while thread j is stuck in its entry protocol, thread i infinitely

Table 1 Verification results.

Algorithm	*# threads*	*Safe*	*Regular*	*Atomic*			
		T	*T*	*T*	*S*	*I*	*A*
Anderson [4]	2	S	S	S	S	M	M
Aravind BLRU [6]	3	S	S	S	M	M	M
Aravind BLRU (alt.)	3	S	S	S	S	M	M
Attiya-Welch (orig.) [9]	2	D	S	S	D	M	M
Attiya-Welch (orig., alt.)	2	S	S	S	D	M	M
Attiya-Welch (var.) [49]	2	M	M	S	D	M	M
Attiya-Welch (var., alt.)	2	S	S	S	D	M	M
Burns-Lynch [17]	3	D	D	D	D	M	M
Dekker [3]	2	M	M	S	D	M	M
Dekker (alt.)	2	M	M	S	S	M	M
Dekker RW-safe [15]	2	S	S	S	D	M	M
Dekker RW-safe (DFtoSF)	2	S	S	S	S	M	M
Dijkstra [20]	3	M	D	D	M	M	M
Kessels [34]	2	X	X	S	S	M	M
Knuth [35]	3	M	S	S	M	M	M
Lamport 1-bit [38]	3	D	D	D	D	M	M
Lamport 1-bit (DFtoSF)	3	S	S	S	S	M	M
Lamport 3-bit [38]	3	S	S	S	S	M	M
Peterson [46]	2	X	X	S	S	M	M
Szymanski flag (int.) [52]	3	X	X	S	S	M	M
Szymanski flag (bit) [52]	3	X	X	X	X	X	X
Szymanski 3-bit lin. wait [53]	3	X	X	X	X	X	X
Szymanski 3-bit lin. wait (alt.)	2	S	S	S	S	M	M

Algorithm 1 Peterson's algorithm.

1: $flag[i] \leftarrow true$
2: $turn \leftarrow i$
3: **await** $flag[j] = false \vee turn = j$
4: **critical section**
5: $flag[i] \leftarrow false$

Algorithm 2 Dekker's algorithm.

1: $flag[i] \leftarrow true$
2: **while** $flag[j] = true$ **do**
3: **if** $turn = j$ **then**
4: $flag[i] \leftarrow false$
5: **await** $turn = i$
6: $flag[i] \leftarrow true$
7: **critical section**
8: $turn \leftarrow j$
9: $flag[i] \leftarrow false$

Algorithm 3 Aravind's BLRU algorithm.

1: $flag[i] \leftarrow true$
2: **repeat**
3: $stage[i] \leftarrow false$
4: **await** $\forall_{j \neq i} : flag[j] = false \vee date[i] < date[j]$
5: $stage[i] \leftarrow true$
6: **until** $\forall_{j \neq i} : stage[j] = false$
7: **critical section**
8: $date[i] \leftarrow \max(date[0], ..., date[N-1]) + 1$
9: **if** $date[i] \geq 2N - 1$ **then**
10: $\forall_{j \in [0 \ldots N-1]} : date[j] \leftarrow j$
11: $stage[i] \leftarrow false$
12: $flag[i] \leftarrow false$

often accesses the critical section. While we do not (yet) have an argument that deadlock freedom is impossible to satisfy if reads can block writes for all possible algorithms, we do observe this to be the case for all algorithms we have analysed.

For many algorithms, it is possible for both competing threads to become stuck in their entry protocol. Consider, for example, Peterson's algorithm from [46], here given as Algorithm 1. If *turn* is initially 0, and thread 1 manages to set *flag*[1] to *true* before thread 0 starts the competition, then on line 3 thread 0 will get stuck in a busy waiting loop. Thread 1 needs to set *turn* to 1 to let thread 0 pass line 3, but thread 0's repeated reads of *turn* prevent this write from taking place, resulting in both threads being trapped in the entry protocol. An alternative way to get a deadlock freedom violation is via the exit protocol. Once a thread has finished its critical section access, it needs to communicate that it no longer requires access to the other thread. In Peterson's, this is done on line 5 by setting the thread's *flag* to *false*. However, if the other thread is repeatedly reading this register, such as is done on line 3, then the completion of the exit protocol can be blocked, once again preventing both threads from accessing their critical sections.

We see similar behaviour in all algorithms we analyse. Frequently, although not always, the problem lies in busy waiting loops. Given this behaviour, it would be interesting to modify our models to treat busy waiting reads differently from normal reads, and only allow normal reads to interfere with writes. This would give us greater insight into whether for some of the algorithms it is truly the busy waiting that is the source of the deadlock freedom violation. We leave this as future work.

6.2 Aravind's BLRU algorithm

Aravind's BLRU algorithm [6], here given as Algorithm 3, is designed for an arbitrary number of threads N. Every thread has three registers: *flag* and *stage*, Booleans that are initialised at *false*, and a natural number *date*, initialised at the thread's own id. We observe that this algorithm satisfies all three properties with safe and regular registers, as claimed in [6]. However, with atomic registers, deadlock freedom is violated under $JA_{\mathcal{B}}^{\bullet s}$. The following

execution for two threads demonstrates this violation:

- Thread 1 moves through lines 1 through 5, setting *flag*[1] and *stage*[1] to *true*. Note that thread 1 can go through line 4 because *flag*[0] = *false*.
- Thread 0 can similarly move through lines 1 through 5; while *flag*[1] = *true*, we do have that *date*[0] = 0 < 1 = *date*[1], so it can pass through line 4. However, on line 6, thread 0 observes *stage*[1] = *true*, so it has to return to line 2.

At this point, thread 0 can repeat lines 2 through 5 endlessly, as long as thread 1 does not set *stage*[1] to *false*. Note that the resulting infinite execution satisfies $JA_{\mathcal{B}}^{\smile\bullet S}$: thread 1's read of *stage*[0], which it has to perform on line 6, is repeatedly blocked by thread 0's writes to *stage*[0] on lines 3 and 5.

This violation can easily be fixed by preventing a thread from endlessly repeating the loop in the entry protocol while the other thread's *stage* is *false*. This can be done by altering line 4 to instead say **await** $\forall_{j \neq i} : \mathit{flag}[j] = \mathit{false} \vee (\mathit{date}[i] < \mathit{date}[j] \wedge \mathit{stage}[j] = \mathit{false})$. While this makes it more difficult to progress through line 4, it is impossible for all threads to get stuck there: if all threads are on line 4, then *stage* is *false* for all of them, and so the one with the lowest *date* can go to line 5. Indeed, as is shown in Table 1, with this modification Aravind's algorithm now satisfies starvation freedom under $JA_{\mathcal{B}}^{\smile\bullet S}$.

6.3 Dekker's algorithm

Dekker's algorithm originally appears in [19]. There is no clear pseudocode given there, so we use the pseudocode from [3], here given as Algorithm 2. The algorithm uses a Boolean *flag* per thread, initially *false*, and a multi-writer register *turn*, initially 0. An execution showing that Dekker's algorithm does not satisfy starvation freedom with safe registers is reported in [15]. This same execution can be found by mCRL2:

- Thread 0 goes through the algorithm without competition, and starts setting *flag*[0] to *false* in the exit protocol, when it is currently *true*.
- Thread 1 starts the competition and reads *flag*[0] = *false*, the new value, on line 2. It can therefore go to the critical section, and set *turn* to 0 in the exit protocol.
- Thread 1 then starts the competition again, now reading *flag*[0] = *true*, the old value, on line 2. Since *turn* = 0, it goes to line 5 and starts waiting for *turn* to be 1.
- Thread 0 finishes the exit protocol and never re-attempts to enter the critical section.

Since thread 0 will never set *turn* to 1, thread 1 can never escape line 5. This execution violates deadlock freedom as well as starvation freedom, and is also applicable to regular registers. The phenomenon where two reads concurrent with the same write return first the new and then the old value is called *new-old inversion*, and is explicitly allowed by Lamport in his definitions of safe and regular registers [40]. An interesting quality of this execution is that it relies on thread 0 only finitely often executing the algorithm. If we did not define the actions nc_t for all $t \in \mathbb{T}$ to be blockable, this execution would be missed.

In [15] the following improvements are suggested to make the algorithm "RW-safe", i.e., correct with safe registers: on line 5, **await** *turn* = $i \vee$ *flag*[j] = *false*, and on line 8, only write to *turn* if its value would be changed. Our model checking confirms that with these alterations, starvation freedom is satisfied with both safe and regular registers.

In [45], it is claimed that Dekker's algorithm without alterations is correct with non-atomic registers. Instead of dealing with the spurious violations of liveness properties via completeness criteria, they use the model checking tool UPPAAL [10] to compute the maximum number of times a thread may be overtaken by another thread. They determine that bound to be finite and conclude starvation freedom is satisfied. However, the deadlock

freedom violation observed here and in [15] shows a thread never gaining access to the critical section while only being overtaken once. Hence, finding a finite upper bound to the number of overtakes is insufficient to establish deadlock freedom.

As can be observed in Table 1, both the version of Dekker's algorithm presented in [3] and the RW-safe version from [15] are starvation-free with atomic registers under $JA_{\mathcal{B}}^{\smile\bullet T}$, but only deadlock-free under $JA_{\mathcal{B}}^{\smile\bullet S}$. In both variants, this is because one thread, say 0, can remain stuck on line 5 trying to perform a read, while the other thread, in this case 1, repeatedly executes the full algorithm without having to wait on thread 0, since $\mathit{flag}[0] = \mathit{false}$. In the process, thread 1 writes to the variable that thread 0 is trying to read, meaning this execution is just under $JA_{\mathcal{B}}^{\smile\bullet S}$.

This starvation freedom violation can be easily fixed for the presentation in [3], since the variable that thread 0 is trying to read on line 5 is *turn*. If we alter the algorithm so that a thread only writes to *turn* on line 8 if this would change the value, then starvation freedom is satisfied. This change is part of the changes suggested in [15], yet that version of the algorithm is not starvation-free under $JA_{\mathcal{B}}^{\smile\bullet S}$. This is due to the other change: on line 5, thread 0 now also has to read $\mathit{flag}[1]$, the value of which thread 1 does change every time it executes the algorithm. This violation therefore cannot be fixed so easily.

6.4 Szymanski's 3-bit linear wait algorithm

Algorithm 4 Szymanski's 3-bit linear wait algorithm.

```
1:  a[i] ← true
2:  for j from 0 to N−1 do await s[j] = false
3:  w[i] ← true
4:  a[i] ← false
5:  while s[i] = false do
6:      j ← 0
7:      while j < N ∧ a[j] = false do j ← j + 1
8:      if j = N then
9:          s[i] ← true
10:         j ← 0
11:         while j < N ∧ a[j] = false do j ← j + 1
12:         if j < N then s[i] ← false
13:         else
14:             w[i] ← false
15:             for j from 0 to N − 1 do await w[j] = false
16:     if j < N then
17:         j ← 0
18:         while j < N ∧ (w[j] = true ∨ s[j] = false) do j ← j + 1
19:     if j ≠ i ∧ j < N then
20:         s[i] ← true
21:         w[i] ← false
22: for j from 0 to i − 1 do await s[j] = false
23: critical section
24: s[i] ← false
```

In [53], Szymanski proposes four mutual exclusion algorithms. Here, we discuss the first: the 3-bit linear wait algorithm, which is claimed to be correct with non-atomic registers. See Algorithm 4. Each thread has three Booleans, a, w, and s, all initially *false*. An execution showing a mutual exclusion violation for safe, regular, and atomic registers with three threads for the 3-bit linear wait algorithm is given in [50].

With two threads, we do still get a mutual exclusion violation with safe and regular registers, given in detail in [50], but not with atomic registers. The violation specifically relies on reading $w[j]$ before $s[j]$ on line 18, so that there can be new-old inversion on $s[j]$, while still obtaining the value of $w[j]$ from before the other thread started writing to $s[j]$. We therefore considered the alternative, where $s[j]$ is read before $w[j]$ on line 18. With this change and only two threads, the algorithm satisfies all expected properties.

The observation that Szymanski's 3-bit linear wait algorithm violates mutual exclusion with three threads is also made in [44]. They suggest a different way to make the algorithm correct for two threads: instead of swapping the reads on line 18, they require a thread to also read $w[j]$ on line 22. We did not investigate this suggested change further, as we prefer the solution that does not require an additional read.

6.5 From deadlock freedom to starvation freedom

In [48, Section 2.2.2], an algorithm is presented to turn any mutual exclusion algorithm that satisfies mutual exclusion and deadlock freedom into one that satisfies starvation freedom as well. It is due to Yoah Bar-David (1998) and first appears in [54]. We take the pseudocode from [26], where it is proven that this algorithm works for safe, regular and atomic registers.

To confirm the correctness of the algorithm experimentally, we applied it to Lamport's 1-bit algorithm [38], which (by design) does not satisfy starvation freedom at all, and to the RW-safe version of Dekker's algorithm [15], where starvation freedom only fails due to writes interfering with reads. The results are given in the table with "DFtoSF". We indeed find that starvation freedom is now satisfied where previously only deadlock freedom was.

7 Related work

To the best of our knowledge, we are the first to do automatic verification of mutual exclusion algorithms while incorporating both which operations can block each other and the effects of overlapping write operations. The two elements have been considered separately previously.

In [14], starvation-freedom of Peterson's algorithm with atomic registers is checked using mCRL2 under the justness assumption. Two different concurrency relations are considered, which are similar to our $\smile\!\!\bullet_S$ and $\smile\!\!\bullet_A$.

There have been many formal verifications of mutual exclusion algorithms with atomic registers [11, 43, 29]. Non-atomic registers have been covered less frequently, and their verification is often restricted to single-writer registers [41, 15]. The safety properties of mutual exclusion algorithms with MWMR non-atomic registers were verified using mCRL2 in [50]. In several papers by Nigro, including [44, 45], safety and liveness verification with MWMR non-atomic registers has been done using UPPAAL.

A major drawback of our approach is that we only consider a small number of parallel threads. There has been much work in the literature on parametrised verification, where the correctness of an algorithm is established for an arbitrary number of threads [13, 22, 55, 12]. Where these techniques handle liveness, it is under forms of weak or strong fairness, not justness. To our knowledge, these techniques have not yet been applied in the context

of non-atomic registers. While several papers on parametrised verification, such as [2, 1], mention dropping the atomicity assumption, this refers to evaluating existential and universal conditions on all threads in a single atomic operation, rather than atomicity of the registers.

In [7, 31, 32] and other work by Hesselink, interactive theorem proving with the proof assistant PVS is used to check safety and liveness properties (under fairness) of mutual exclusion algorithms for an arbitrary number of threads with non-atomic registers. To our knowledge, the case of writes overlapping each other has not been covered with this technique.

8 Conclusion

When it comes to analysing the correctness of algorithms, particularly when considering non-atomic registers, behavioural reasoning is often insufficient. Mistakes can be subtle, and may depend on edge-cases that are easily overlooked. Model checking is a solution here; we formally model the threads executing the algorithm, as well as the registers through which they communicate, and the entire state-space is searched for possibly violations of correctness properties. In this work, we verify a large number of mutual exclusion algorithms using the model checking toolset mCRL2. We expand on previous work by checking liveness properties – deadlock freedom and starvation freedom – in addition to the main safety property. To circumvent spurious violating paths in our models, we incorporate the completeness criterion justness into our verifications. We checked algorithms under six different memory models, where a memory model is a combination of a register model (safe, regular, or atomic) and a model of which register access operations can block each other. The former dimension we capture in the models themselves, by modelling the behaviour of the three types of register. The latter, we capture in the concurrency relations employed as part of justness. We found a number of interesting violations of correctness properties, and in some cases could suggest improvements to algorithms to fix these violations. We find that there are several algorithms that satisfy all three properties for four out of six memory models. For three threads, this is accomplished by the fixed version of Aravind's BLRU algorithm and Lamport's 3-bit algorithm. If we also consider algorithms for just two threads, then Anderson's algorithm and the fixed version of Szymanski's 3-bit linear wait algorithm also meet this bar. We also considered an algorithm to turn deadlock-free algorithms into starvation-free ones, and experimentally confirmed that it indeed works for Dekker's algorithm made RW-safe and Lamport's 1-bit algorithm.

References

1 Parosh Aziz Abdulla, Frédéric Haziza, and Lukáš Holík. Parameterized verification through view abstraction. *International Journal on Software Tools for Technology Transfer*, 18(5):495–516, 2016. doi:10.1007/s10009-015-0406-x.

2 Parosh Aziz Abdulla, Noomene Ben Henda, Giorgio Delzanno, and Ahmed Rezine. Handling parameterized systems with non-atomic global conditions. In Francesco Logozzo, Doron A. Peled, and Lenore D. Zuck, editors, *Verification, Model Checking, and Abstract Interpretation* (VMCAI'08), volume 4905 of *Lecture Notes in Computer Science*, pages 22–36. Springer, 2008. doi:10.1007/978-3-540-78163-9_7.

3 K. Alagarsamy. Some myths about famous mutual exclusion algorithms. *SIGACT News*, 34(3):94–103, 2003. doi:10.1145/945526.945527.

4 James H. Anderson. A fine-grained solution to the mutual exclusion problem. *Acta Informatica*, 30(3):249–265, 1993. doi:10.1007/BF01179373.

5 Krzysztof R. Apt and Ernst-Rüdiger Olderog. Proof rules and transformations dealing with fairness. *Science of Computer Programming*, 3(1):65–100, 1983. doi:10.1016/0167-6423(83)90004-7.

6 Alex A. Aravind. Yet another simple solution for the concurrent programming control problem. *IEEE Transactions on Parallel and Distributed Systems*, 22(6):1056–1063, 2011. doi:10.1109/TPDS.2010.172.

7 Alex A. Aravind and Wim H. Hesselink. Nonatomic dual bakery algorithm with bounded tokens. *Acta Informatica*, 48(2):67–96, 2011. doi:10.1007/s00236-011-0132-0.

8 Hagit Attiya, Rachid Guerraoui, Danny Hendler, Petr Kuznetsov, Maged M. Michael, and Martin T. Vechev. Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. In Thomas Ball and Mooly Sagiv, editors, Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on *Principles of Programming Languages* (POPL'11), pages 487–498. ACM, 2011. doi:10.1145/1926385.1926442.

9 Hagit Attiya and Jennifer L. Welch. *Distributed computing - fundamentals, simulations, and advanced topics* (2nd ed.). Wiley series on parallel and distributed computing. Wiley, 2004.

10 Gerd Behrmann, Alexandre David, and Kim Guldstrand Larsen. A tutorial on uppaal. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems,* International School on Formal Methods for the Design of Computer, Communication and Software Systems, (SFM-RT'04), volume 3185 of *Lecture Notes in Computer Science*, pages 200–236. Springer, 2004. doi:10.1007/978-3-540-30080-9_7.

11 Mordechai Ben-Ari. *Principles of the Spin model checker*. Springer, 2008. doi:10.1007/978-1-84628-770-1.

12 Roderick Bloem, Swen Jacobs, Ayrat Khalimov, Igor Konnov, Sasha Rubin, Helmut Veith, and Josef Widder. Decidability in parameterized verification. *SIGACT News*, 47(2):53–64, 2016. doi:10.1145/2951860.2951873.

13 Ahmed Bouajjani, Bengt Jonsson, Marcus Nilsson, and Tayssir Touili. Regular model checking. In E. Allen Emerson and A. Prasad Sistla, editors, *Computer Aided Verification* (CAV'00), volume 1855 of *Lecture Notes in Computer Science*, pages 403–418. Springer, 2000. doi:10.1007/10722167_31.

14 Mark Bouwman, Bas Luttik, and Tim A. C. Willemse. Off-the-shelf automated analysis of liveness properties for just paths. *Acta Informatica*, 57(3-5):551–590, 2020. doi:10.1007/s00236-020-00371-w.

15 Peter A. Buhr, David Dice, and Wim H. Hesselink. Dekker's mutual exclusion algorithm made RW-safe. *Concurrency and Computation: Practice and Experience*, 28(1):144–165, 2016. doi:10.1002/cpe.3659.

16 Olav Bunte, Jan Friso Groote, Jeroen J. A. Keiren, Maurice Laveaux, Thomas Neele, Erik P. de Vink, Wieger Wesselink, Anton Wijs, and Tim A. C. Willemse. The mCRL2 toolset for analysing concurrent systems – improvements in expressivity and usability. In Tomás Vojnar and Lijun Zhang, editors, *Tools and Algorithms for the Construction and Analysis of Systems* (TACAS'19), held as part of the European Joint Conferences on *Theory and Practice of Software* (ETAPS'19), Part II, volume 11428 of *Lecture Notes in Computer Science*, pages 21–39. Springer, 2019. doi:10.1007/978-3-030-17465-1_2.

17 James E. Burns and Nancy A. Lynch. Bounds on shared memory for mutual exclusion. *Information and Computation*, 107(2):171–184, 1993. doi:10.1006/inco.1993.1065.

18 Flavio Corradini, Maria Rita Di Berardini, and Walter Vogler. Time and fairness in a process algebra with non-blocking reading. In Mogens Nielsen, Antonín Kucera, Peter Bro Miltersen, Catuscia Palamidessi, Petr Tuma, and Frank D. Valencia, editors, *SOFSEM'09: Theory and Practice of Computer Science, 35th Conference on Current Trends in Theory and Practice of Computer Science*, volume 5404 of *Lecture Notes in Computer Science*, pages 193–204. Springer, 2009. doi:10.1007/978-3-540-95891-8_20.

19 Edsger W Dijkstra. Over de sequentialiteit van procesbeschrijvingen (ewd-35). *Center for American History, University of Texas at Austin*, 1962. URL: `http://www.cs.utexas.edu/users/EWD/ewd00xx/EWD35.PDF`.

20 Edsger W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965. `doi:10.1145/365559.365617`.

21 Victor Dyseryn, Rob J. van Glabbeek, and Peter Höfner. Analysing mutual exclusion using process algebra with signals. In Kirstin Peters and Simone Tini, editors, Proceedings Combined 24th International Workshop on *Expressiveness in Concurrency* and 14th Workshop on *Structural Operational Semantics* (EXPRESS/SOS'17), volume 255 of *Electronic Proceedings in Theoretical Computer Science*, pages 18–34, 2017. `doi:10.4204/EPTCS.255.2`.

22 Dana Fisman and Amir Pnueli. Beyond regular model checking. In Ramesh Hariharan, Madhavan Mukund, and V. Vinay, editors, *Foundations of Software Technology and Theoretical Computer Science* (FST&TCS'01), volume 2245 of *Lecture Notes in Computer Science*, pages 156–170. Springer, 2001. `doi:10.1007/3-540-45294-X_14`.

23 Rob J. van Glabbeek. Is speed-independent mutual exclusion implementable? (Invited talk). In Sven Schewe and Lijun Zhang, editors, 29th International Conference on *Concurrency Theory (CONCUR'18)*, volume 118 of *LIPIcs*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018. `doi:10.4230/LIPIcs.CONCUR.2018.3`.

24 Rob J. van Glabbeek. Justness - A completeness criterion for capturing liveness properties (extended abstract). In Mikolaj Bojanczyk and Alex Simpson, editors, *Foundations of Software Science and Computation Structures* (FOSSACS'19), held as part of the European Joint Conferences on *Theory and Practice of Software* (ETAPS'19), volume 11425 of *Lecture Notes in Computer Science*, pages 505–522. Springer, 2019. `doi:10.1007/978-3-030-17127-8_29`.

25 Rob J. van Glabbeek. Modelling mutual exclusion in a process algebra with time-outs. *Information and Computation*, 294:105079, 2023. `doi:10.1016/j.ic.2023.105079`.

26 Rob J. van Glabbeek and Daniele Gorla. On the notions of bounded bypass, and how to make any deadlock-free mutex protocol satisfy one of them, 2025. To appear.

27 Rob J. van Glabbeek and Peter Höfner. Progress, justness, and fairness. *ACM Computing Surveys*, 52(4), 2019. `doi:10.1145/3329125`.

28 Rob J. van Glabbeek, Bas Luttik, and Myrthe S.C. Spronck. Just verification of mutual exclusion algorithms, 2025. Full version of the present paper. `doi:10.48550/arXiv.2507.13198`.

29 Jan Friso Groote and Jeroen J. A. Keiren. Tutorial: Designing distributed software in mCRL2. In Kirstin Peters and Tim A. C. Willemse, editors, *Formal Techniques for Distributed Objects, Components, and Systems - 41st IFIP WG 6.1 International Conference, FORTE 2021, Held as Part of the 16th International Federated Conference on Distributed Computing Techniques, DisCoTec 2021, Valletta, Malta, June 14-18, 2021, Proceedings*, volume 12719, pages 226–243, Cham, 2021. Springer. `doi:10.1007/978-3-030-78089-0_15`.

30 Jan Friso Groote and Mohammad Reza Mousavi. *Modeling and Analysis of Communicating Systems*. MIT Press, 2014. `doi:10.7551/mitpress/9946.001.0001`.

31 Wim H. Hesselink. Mechanical verification of lamport's bakery algorithm. *Science of Computer Programming*, 78(9):1622–1638, 2013. `doi:10.1016/j.scico.2013.03.003`.

32 Wim H. Hesselink. Mutual exclusion by four shared bits with not more than quadratic complexity. *Science of Computer Programming*, 102:57–75, 2015. `doi:10.1016/j.scico.2015.01.001`.

33 C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, 1985.

34 Joep L. W. Kessels. Arbitration without common modifiable variables. *Acta Informatica*, 17:135–141, 1982. `doi:10.1007/BF00288966`.

35 Donald E. Knuth. Additional comments on a problem in concurrent programming control. *Communications of the ACM*, 9(5):321–322, 1966. `doi:10.1145/355592.365595`.

36 Leslie Lamport. A new solution of dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, 1974. `doi:10.1145/361082.361093`.

37 Leslie Lamport. The mutual exclusion problem: Part I—a theory of interprocess communication. *J. ACM*, 33(2):313–326, 1986. doi:10.1145/5383.5384.

38 Leslie Lamport. The mutual exclusion problem: Part II—statement and solutions. *J. ACM*, 33(2):327–348, 1986. doi:10.1145/5383.5385.

39 Leslie Lamport. On interprocess communication. Part I: basic formalism. *Distributed Comput.*, 1(2):77–85, 1986. doi:10.1007/BF01786227.

40 Leslie Lamport. On interprocess communication. Part II: algorithms. *Distributed Comput.*, 1(2):86–101, 1986. doi:10.1007/BF01786228.

41 Leslie Lamport. The TLA+ Hyperbook, 2015. chapter 7.8.4: The Real Bakery Algorithm. URL: http://lamport.azurewebsites.net/tla/hyperbook.html.

42 Daniel Lehmann, Amir Pnueli, and Jonathan Stavi. Impartiality, justice and fairness: The ethics of concurrent termination. In Shimon Even and Oded Kariv, editors, *Automata, Languages and Programming* (ICALP'81), volume 115 of *Lecture Notes in Computer Science*, pages 264–277. Springer, 1981. doi:10.1007/3-540-10843-2_22.

43 Radu Mateescu and Wendelin Serwe. Model checking and performance evaluation with CADP illustrated on shared-memory mutual exclusion protocols. *Science of Computer Programming*, 78(7):843–861, 2013. Special section on Formal Methods for Industrial Critical Systems (FMICS 2009 + FMICS 2010) & Special section on Object-Oriented Programming and Systems (OOPS 2009), a special track at the 24th ACM Symposium on Applied Computing. doi:10.1016/j.scico.2012.01.003.

44 Libero Nigro. Verifying mutual exclusion algorithms with non-atomic registers. *Algorithms*, 17(12), 2024. doi:10.3390/a17120536.

45 Libero Nigro, Franco Cicirelli, and Francesco Pupo. Modeling and analysis of Dekker-based mutual exclusion algorithms. *Computers*, 13(6):133, 2024. doi:10.3390/computers13060133.

46 Gary L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, 1981. doi:10.1016/0020-0190(81)90106-X.

47 Gary L. Peterson. A new solution to lamport's concurrent programming problem using small shared variables. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(1):56–65, 1983. doi:10.1145/357195.357199.

48 Michel Raynal. *Concurrent Programming - Algorithms, Principles, and Foundations*. Springer, 2013. doi:10.1007/978-3-642-32027-9.

49 Cheng Shao, Jennifer L. Welch, Evelyn Pierce, and Hyunyoung Lee. Multiwriter consistency conditions for shared memory registers. *SIAM Journal on Computing*, 40(1):28–62, 2011. doi:10.1137/07071158X.

50 Myrthe S. C. Spronck and Bas Luttik. Process-algebraic models of multi-writer multi-reader non-atomic registers. In Guillermo A. Pérez and Jean-François Raskin, editors, 34th International Conference on *Concurrency Theory* (CONCUR'23), volume 279 of *LIPIcs*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. Full version available at https://arxiv.org/abs/2307.05143. doi:10.4230/LIPIcs.CONCUR.2023.5.

51 Myrthe S. C. Spronck, Bas Luttik, and Tim A. C. Willemse. Progress, justness and fairness in modal μ-calculus formulae. In Rupak Majumdar and Alexandra Silva, editors, 35th International Conference on *Concurrency Theory* (CONCUR'24), volume 311 of *LIPIcs*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPIcs.CONCUR.2024.38.

52 Boleslaw K. Szymanski. A simple solution to lamport's concurrent programming problem with linear wait. In Jacques Lenfant, editor, Proceedings of the 2nd international conference on *Supercomputing* (ICS'88), pages 621–626. ACM, 1988. doi:10.1145/55364.55425.

53 Boleslaw K. Szymanski. Mutual exclusion revisited. In Joshua Maor and Abraham Peled, editors, *Next Decade in Information Technology:* Proceedings of the 5th *Jerusalem Conference on Information Technology*, pages 110–117. IEEE Computer Society, 1990. doi:10.1109/JCIT.1990.128275.

54 Gadi Taubenfeld. *Synchronization algorithms and concurrent programming*. Pearson Education, 2006.

55 Lenore D. Zuck and Amir Pnueli. Model checking and abstraction to the aid of parameterized systems (a survey). *Computer Languages, Systems & Structures*, 30(3):139–169, 2004. doi:10.1016/j.cl.2004.02.006.

A Register models

In this appendix, we expand on the material of Section 3, by giving the formal process-algebraic models of the registers. First, we present the general structure of such a model and explain how to translate it to an LTS. We then give the three models, as well as the definitions of all the access and update functions. We leave the status object abstract: it is not necessary to define the data structure itself, as long as it is clear what information can be retrieved from it.

A.1 Structure and functions

Recall that we use $\mathbb{T}$ for the thread identifiers, $\mathbb{R}$ for register identifiers, and that for every $r \in \mathbb{R}$, the set $\mathbb{D}_r$ contains all values that r may hold. Additionally, we use a status object as the finite memory of a register, the set of possible statuses being $\mathbb{S}$.

We define the following structure, shared by all three register models. Let $\gamma \in \{saf, reg, ato\}$, then each model looks as follows, for some number n:

$$Reg_\gamma(r:\mathbb{R}, s:\mathbb{S}) = \sum_{t\in\mathbb{T}} \sum_{d\in\mathbb{D}_r} \sum_{0\le j<n} (c_j(s,t,d) \to a_j(t,d) \cdot Reg_\gamma(r, u_j(s,t,d))) \tag{1}$$

This represents a register with id r, that tracks its status with s. The process first sums over $t \in \mathbb{T}$ and $d \in \mathbb{D}_r$, allowing interaction by all threads and with all possible data parameters. Furthermore, it has n summands, each of the form $c_j(s,t,d) \to a_j(t,d) \cdot Reg_\gamma(r, u_j(s,t,d))$ where $c_j(s,t,d)$ is a Boolean condition, $a_j(t,d)$ is an action, and u_j is an *update function* that takes s, t and d and returns the updated status object $s' \in \mathbb{S}$. Such a process equation gives rise to an LTS. Given a predefined initial state $init \in \mathbb{S}$, the LTS of $Reg_\gamma(r, init)$ is:

$$(\mathbb{S}, \bigcup_{0\le j<n} \{a_j(t,d) \mid t\in\mathbb{T}, d\in\mathbb{D}_r\}, init, \bigcup_{0\le j<n} \{(s, a_j(t,d), u_j(s,t,d)) \mid t\in\mathbb{T}, d\in\mathbb{D}_r \wedge c_j(s,t,d)\}) \tag{2}$$

We now describe the initial state and the update functions. As stated above, we do not wish the go into the implementation details of the status object. Instead, we define a collection of *access functions* which retrieve information from the current state. The initial state, as well as the effects of the update functions, are then defined by how they alter the results of the access functions. We use the following access functions, which are local to any given register r:

- $stor : \mathbb{S} \to \mathbb{D}_r$, the value that is currently stored in the register.
- $rds : \mathbb{S} \to 2^\mathbb{T}$, the set of thread id's of threads that have invoked a read operation on this register that has not yet had its response.
- $wrts : \mathbb{S} \to 2^\mathbb{T}$, the set of thread id's of threads that have invoked a write operation on this register that has not yet had its response.
- $pend : \mathbb{S} \to 2^\mathbb{T}$, the set of thread id's of threads that have invoked an operation that has not been ordered yet. Only used by the regular and atomic models.
- $rec : \mathbb{S} \times \mathbb{T} \to \mathbb{D}_r$, a mapping that allows us to record a single data value per thread, for instance used to remember what value was passed with a start write action.
- The predicate $ovrl$ on $\mathbb{S} \times \mathbb{T}$, which stores whether an ongoing read or write operation of a thread has encountered an overlapping write. Only used by the safe model.

- $posv : \mathbb{S} \times \mathbb{T} \to 2^{\mathbb{D}_r}$, a mapping that stores a set of values per thread, representing the possible return values of an ongoing read by a thread. Only used by the regular model.

The initial state *init* is defined as follows, for all $t \in \mathbb{T}$ and a pre-defined initial value d_{init}:

$$\begin{array}{llll} stor(init) = d_{init} & wrts(init) = \emptyset & rec(init, t) = d_{init} & posv(init, t) = \emptyset \\ rds(init) = \emptyset & pend(init) = \emptyset & ovrl(init, t) = false & \end{array}$$

The initial value d_{init} of a register depends on the modelled algorithm.

We now define the update functions in a similar way to the initial state: by showing how the return values of the access functions are altered by the update function. Each update function corresponds to an action and is applied when that action occurs; if the action's name is a, the update function is called ua. Not every update function uses the data parameter that is passed to it according to (1); in these cases we only give the thread id and status parameters. If an access function is not mentioned, then its return value after the update is the same as before. Given an arbitrary state $s \in \mathbb{S}$, thread id $t \in \mathbb{T}$ and data value $d \in \mathbb{D}_r$:
If $s' = usr(s, t)$, then:

$$\begin{array}{ll} rds(s') = rds(s) \cup \{t\} & ovrl(s', t) = (wrts(s) > 0) \\ pend(s') = pend(s) \cup \{t\} & posv(s', t) = \{stor(s)\} \cup \{d' \mid \exists_{t' \in wrts(s)}. rec(s, t') = d'\} \end{array}$$

If $s' = ufr(s, t)$, then:

$$rds(s') = rds(s) \setminus \{t\}$$

If $s' = usw(s, t, d)$, then for all $t' \neq t$:

$$\begin{array}{lll} wrts(s') = wrts(s) \cup \{t\} & rec(s', t) = d & ovrl(s', t) = (wrts(s) > 0) \\ pend(s') = pend(s) \cup \{t\} & posv(s', t') = posv(s, t') \cup \{d\} & ovrl(s', t') = true \end{array}$$

If $s' = ufw(s, t, d)$, then:

$$stor(s') = d \qquad wrts(s') = wrts(s) \setminus \{t\}$$

If $s' = uor(s, t)$, then:

$$pend(s') = pend(s) \setminus \{t\} \qquad rec(s', t) = stor(s)$$

If $s' = uow(s, t, d)$, then:

$$stor(s') = d \qquad pend(s') = pend(s) \setminus \{t\}$$

These formal definitions correspond to the intuitive descriptions given above. Of note is that $ovrl(s', t')$ is set to *true* whenever a thread $t \neq t'$ starts a write, even if t' is not actively reading or writing. This is done for simplicity of the definition: when t' starts reading or writing, it will reset its own *ovrl* to the correct value, depending on whether there is an overlapping write active at that point. Something similar is done with *posv*: when a write is started, its value gets added to the *posv* sets of every other thread, even if they are not actively reading. When a thread starts reading, it sets its own *posv* correctly.

We now give the three register models.

A.2 Safe MWMR registers

See Figure 1 for the process equation representing our safe register model.

The correspondence between the process and the four rules given for MWMR safe registers in Subsection 3.1 is rather direct: the first two summands allow a thread that is not currently reading or writing to begin a read or a write, and the remaining four each represent one of the four rules, in order. Note that, in the case of a finish write without overlap, we use *rec* to retrieve which value this thread intended to write so that the register state can be appropriately updated.

$$Reg_{saf}(r:\mathbb{R}, s:\mathbb{S}) = \sum_{t\in\mathbb{T}}\sum_{d\in\mathbb{D}_r}\left(\begin{array}{ll} & (t\notin(rds(s)\cup wrts(s))) \to sr_{t,r}\cdot Reg_{saf}(r, usr(s,t)) \\ + & (t\notin(rds(s)\cup wrts(s))) \to sw_{t,r}(d)\cdot Reg_{saf}(r, usw(s,t,d)) \\ + & (t\in rds(s)\wedge\neg ovrl(s,t)) \to fr_{t,r}(stor(s))\cdot Reg_{saf}(r, ufr(s,t)) \\ + & (t\in rds(s)\wedge ovrl(s,t)) \to fr_{t,r}(d)\cdot Reg_{saf}(r, ufr(s,t)) \\ + & (t\in wrts(s)\wedge\neg ovrl(s,t)) \to fw_{t,r}\cdot Reg_{saf}(r, ufw(s,t,rec(s,t))) \\ + & (t\in wrts(s)\wedge ovrl(s,t)) \to fw_{t,r}\cdot Reg_{saf}(r, ufw(s,t,d)) \end{array}\right)$$

Figure 1 Safe register process.

A.3 Regular MWMR registers

See Figure 2 for the process equation representing our regular register model. Recall that regular registers use the order write action to generate a global ordering on all write operations on a register on the fly.

$$Reg_{reg}(r:\mathbb{R}, s:\mathbb{S}) = \sum_{t\in\mathbb{T}}\sum_{d\in\mathbb{D}_r}\left(\begin{array}{ll} & (t\notin(rds(s)\cup wrts(s))) \to sr_{t,r}\cdot Reg_{reg}(r, usr(s,t)) \\ + & (t\notin(rds(s)\cup wrts(s))) \to sw_{t,r}(d)\cdot Reg_{reg}(r, usw(s,t,d)) \\ + & (t\in rds(s)\wedge d\in posv(s,t)) \to fr_{t,r}(d)\cdot Reg_{reg}(r, ufr(s,t)) \\ + & (t\in wrts(s)\wedge t\in pend(s)) \to ow_{t,r}\cdot Reg_{reg}(r, uow(s,t,rec(s,t))) \\ + & (t\in wrts(s)\wedge t\notin pend(s)) \to fw_{t,r}\cdot Reg_{reg}(r, ufw(s,t,stor(s))) \end{array}\right)$$

Figure 2 Regular register process.

Similar to the safe register process, the first two summands are merely allowing an idle thread to begin a read or write operation. The third summand corresponds to finishing a read by returning a value that is in the set of possible values to be returned for this read. Recall that, by the definition of *posv*, this set is constructed as follows: when a read starts, the set is initialised to the current stored value of the register and the intended write value of every active write. Subsequently, whenever a write occurs, its intended value is added to the set. This way, at the finish read, the set will contain exactly those values that the read could return. The fourth summand allows the occurrence of the ordering action; at this time the intended value of the write, which was temporary stored in the access function *rec*, is logged as the stored value of the register. The final summand describes the ending of a write operation. The *ufw* update function will set the stored value to whatever data value is passed. In this case, the stored value should not change at the finish write, since it was already changed at the order write. Hence, we simply pass $stor(s)$. Note that we use *pend* to determine if the order write action still has to occur.

A.4 Atomic MWMR registers

See Figure 3 for our model of MWMR atomic registers. Recall that, in addition to the order write action, the atomic register model also uses the order read action. This way, it generates an ordering on all operations on a register.

$$Reg_{ato}(r:\mathbb{R}, s:\mathbb{S}) = \sum_{t\in\mathbb{T}}\sum_{d\in\mathbb{D}_r}\left(\begin{array}{ll} & (t\notin(rds(s)\cup wrts(s)))\to sr_{t,r}\cdot Reg_{ato}(r, usr(s,t))\\ + & (t\notin(rds(s)\cup wrts(s)))\to sw_{t,r}(d)\cdot Reg_{ato}(r, usw(s,t,d))\\ + & (t\in rds(s)\wedge t\in pend(s))\to or_{t,r}\cdot Reg_{ato}(r, uor(s,t))\\ + & (t\in wrts(s)\wedge t\in pend(s))\to ow_{t,r}\cdot Reg_{ato}(r, uow(s,t,rec(s,t)))\\ + & (t\in rds(s)\wedge t\notin pend(s))\to fr_{t,r}(rec(s,t))\cdot Reg_{ato}(r, ufr(s,t))\\ + & (t\in wrts(s)\wedge t\notin pend(s))\to fw_{t,r}\cdot Reg_{ato}(r, ufw(s,t,stor(s))) \end{array}\right)$$

Figure 3 Atomic register process.

Summands one, three and five are the invocation, ordering and response of a read operation respectively. Similarly, summands two, four and six are the invocation, ordering and response of a write operation. We use *pend* to determine whether an operation's order action still has to occur. When a read is ordered, we save the current stored value of the register in *rec*, so that this value can be returned when the read ends. For writes, we already update *stor* when the write is ordered, meaning that when the write ends we do not want to change *stor* further and just pass the current value back.

B Thread consistency proof

In the [28, Appendix C], we provide a detailed proof of Lemma 6. Here, we give a less detailed version.

▶ **Lemma 6.** Let $M = (\mathcal{S}, Act, init, Trans, thr, reg)$ be a thread-register model. Then the LTS $(\mathcal{S}, Act, init, Trans)$ is thread consistent with respect to the mapping thr.

Proof. Consider a thread-register model $M = (\mathcal{S}, Act, init, Trans, thr, reg)$ and a specific state $s \in \mathcal{S}$. Let a be an arbitrary action enabled in s and assume there exists a transition $s \xrightarrow{b} s'$ to some $s' \in \mathcal{S}$ such that $thr(a) \neq thr(b)$. We must prove that a is enabled in s'. Let $thr(a) = t$ and $reg(a) = r$. Note that a is either a thread local action, or a register (local or interface) action. We consider both cases.

- If a is a thread local action, then it is enabled whenever a is enabled in the LTS belonging to t. Since b is an action by a different thread, the local state of the LTS of t is not affected by b, and so is the same in both s and s'. Thus, a is still enabled in s'.
- If a is not a thread local action, then $r \neq \bot$ and a involves the register r. Note that if $reg(b) \neq r$, then b can affect neither the local state of the LTS of t nor the local state of the LTS of r, so a is guaranteed to still be enabled in s'. We proceed under the assumption that $reg(b) = r$. It is still the case that b cannot affect the local state of the LTS associated with t, so this cannot prevent a from being enabled in s'. It therefore remains to prove that a is enabled in the local state of the LTS associated with r in s'. Let s_r be the local state of the LTS associated with r in s, and s'_r be its equivalent in s'. We proceed by considering which action a is.
  - If a is $sr_{t,r}$, $sw_{t,r}(d)$ for some $d \in \mathbb{D}_r$, $fw_{t,r}$, $or_{t,r}$ or $ow_{t,r}$, then whether it is enabled in state s' is only dependent on t's presence in a subset of $rds(s'_r)$, $wrts(s'_r)$ and $pend(s'_r)$. Adding t to or removing t from these sets can only be done by update functions that belong to actions c with $thr(c) = t$. Since $thr(b) \neq t$, the inclusion of t in these sets is unaltered between s_r and s'_r. Hence, since a is enabled in s_r, it is also enabled in s'_r.

- It remains to consider the case that a is $fr_{t,r}(d)$ for some $d \in \mathbb{D}_r$. We must show that, even if b is an operation on the register r, t can still read the value from r in s' that it could read in s. We do a case distinction on the type of register r.
  - If r is a safe register, then d can be read in s because $ovrl(s_r, t)$ or $stor(s_r) = d$. In the former case, regardless of which action b is, it will still be the case that $ovrl(s'_r, t)$ holds, so t can still return d. In the latter case, while it is possible that b changes the stored value of r if $b = fw_{thr(b),r}$, this means that $thr(b)$ is an active writer while t is an active reader, so there is overlap. Thus, $stor(s'_r) = d$ or $ovrl(s'_r, t)$ is true. Either way, t can still return d in s'.
  - If r is a regular register, then d is in $posv(s_r, t)$. The set of possible values only grows as long as a read is active; it only resets when a new read starts. Since b cannot be a start read by t, d is in $posv(s'_r, t)$, and so t can return d in s'.
  - If r is atomic, then when the finish read action a occurs, it simply returns the value that was previously recorded when the read was ordered. Since $thr(b) \neq t$, $rec(s_r, t) = rec(s'_r, t)$. Therefore t can still return d in s'.

In each case, we have shown that a is enabled in s'. ◀

A State-Based $O(m \log n)$ Partitioning Algorithm for Branching Bisimilarity

Jan Friso Groote
Department of Computer Science, Eindhoven University of Technology, The Netherlands

David N. Jansen
Key Laboratory of System Software, Chinese Academy of Sciences, Beijing, China
State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China

Abstract

We present a new $O(m \log n)$ algorithm to calculate branching bisimulation equivalence, which is the finest commonly used behavioural equivalence on labelled transition systems that takes the internal action τ into account. This algorithm combines the simpler data structure of an earlier algorithm for Kripke structures (without action labels) with the memory-efficiency of a later algorithm partitioning sets of labelled transitions. It employs a particularly elegant four-way split of blocks of states, which refines a block under two splitters and isolates all new bottom states, simultaneously. Benchmark results show that this new algorithm outperforms the best known algorithm for branching bisimulation both in time and space.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms; Software and its engineering → Formal software verification

Keywords and phrases Algorithm, Branching bisimulation, Partition refinement of states

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2025.18

Funding *David N. Jansen*: This work is supported by ISCAS Basic Research ISCAS-JCZD-202302 and is part of the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant no. 101008233.

1 Introduction

Branching bisimulation relates behaviourally equivalent states in labelled transition systems that represent behaviour [4]. It takes internal or invisible steps τ into account and preserves the branching structure of processes. It is slightly finer than weak bisimulation [11].

Almost immediately after the discovery of branching bisimulation a decently efficient $O(mn)$-algorithm (where m is the number of transitions and n is the number of states) became available [7]. This was a great asset not only because it allowed to decide quickly whether states were branching bisimilar, but also because it could be used as an efficient preprocessing step for many other behaviour equivalences that take internal steps into account.

A major improvement was the development of an $O(m \log n)$ algorithm [8, 5], working on Kripke structures, which can be regarded as labelled transition systems where all transitions are labelled with τ, as, at the time, it was not clear how to efficiently deal with multiple transition labels. This algorithm could be applied to arbitrary labelled transition systems by translating them to Kripke structures where one extra state was introduced for every transition in the original transition system [1]. As the number of transitions is generally much larger than the number of states, this is costly, both in time and space.

Following [14], where it was proposed to use equivalence classes of transitions to avoid redoing work that has already been done, a new algorithm was developed that works directly on labelled transition systems [10]. By avoiding the costly translation to Kripke structures, this algorithm was 40% faster than the best algorithm known until then.

© Jan Friso Groote and David N. Jansen;
licensed under Creative Commons License CC-BY 4.0

However, the use of equivalence classes of transitions feels unpleasant, given that there are in general far more transitions than states. So, the purpose of this paper is to design an algorithm that avoids partitioning transitions. The essential step that enables us to do so is the efficient grouping of transitions with the same transition label. Instead of the complex trickery used in previous algorithms, we employ a very natural and elegant four-way split of blocks, which is an extension of the three-way split used in the classical algorithm for strong bisimulation on Kripke structures by Paige and Tarjan [12, 2]. The fourth block collects the infamous "new bottom states", that invalidate the stability invariant, allowing us to isolate the work to repair this invariant, avoiding much of the complexity of earlier algorithms.

The result is a new algorithm that is not only simpler and more elegant, but also faster and more memory-efficient than all its predecessors, although it has the same $O(m \log n)$ time complexity – here we assume that the number of action labels is smaller than the number of transitions. Especially on large transition systems with many transitions a 40% reduction in memory usage and a fourfold reduction in time appears to be possible.

2 Branching Bisimilarity

In this section we define labelled transition systems and branching bisimilarity.

▶ **Definition 2.1** (Labelled transition system)**.** A *labelled transition system* (LTS) is a triple $A = (S, Act, \rightarrow)$ where

1. S is a finite set of *states.* The number of states is denoted by n.
2. Act is a finite set of actions including the *internal action* τ.
3. $\rightarrow \subseteq S \times Act \times S$ is a *transition relation.* The number of transitions m is always finite.

It is common to write $s \xrightarrow{a} s'$ for $(s, a, s') \in \rightarrow$. We write $s \xrightarrow{a} s' \in T$ instead of $(s, a, s') \in T$ for $T \subseteq \rightarrow$. We also write $s \xrightarrow{a} S'$ for the set of transitions $\{s \xrightarrow{a} s' \mid s' \in S'\}$, and likewise $S_1 \xrightarrow{a} S_2$ for the set $\{s_1 \xrightarrow{a} s_2 \mid s_1 \in S_1 \text{ and } s_2 \in S_2\}$. We refer to all actions except τ as the visible actions. The transitions labelled with τ are called invisible. If $s \xrightarrow{a} s'$, we say that from s, the state s', the action a, and the transition $s \xrightarrow{a} s'$ are *reachable.* We write $in(s)$ for the incoming transitions of state s and $out(s)$ for the outgoing transitions of s. Likewise, we write $in(B)$ and $out(B)$ for all incoming and outgoing transitions of a set $B \subseteq S$.

▶ **Definition 2.2** (Branching bisimilarity)**.** Let $A = (S, Act, \rightarrow)$ be a labelled transition system. We call a relation $R \subseteq S \times S$ a *branching bisimulation relation* iff it is symmetric and for all $s, t \in S$ such that $s \mathrel{R} t$ and all transitions $s \xrightarrow{a} s'$ we have:

1. $a = \tau$ and $s' \mathrel{R} t$, or
2. there is a sequence $t \xrightarrow{\tau} \cdots \xrightarrow{\tau} t' \xrightarrow{a} t''$ such that $s \mathrel{R} t'$ and $s' \mathrel{R} t''$.

Two states s and t are *branching bisimilar,* denoted by $s \leftrightarrows_b t$, iff there is a branching bisimulation relation R such that $s \mathrel{R} t$.

If we restrict the definition of branching bisimilarity such that states are only related if they both have, or both have not an infinite sequence of τ's through branching bisimilar states, we obtain the notion of *divergence-preserving* branching bisimulation. This notion is useful as it – contrary to branching bisimilarity – preserves liveness properties. The algorithm that is presented below can be used for divergence-preserving branching bisimilarity with only a minor modification.

Given an equivalence relation R, a transition $s \xrightarrow{a} t$ is called *R-inert* iff $a = \tau$ and $s \mathrel{R} t$. If $t \xrightarrow{\tau} t_1 \xrightarrow{\tau} \cdots \xrightarrow{\tau} t_{n-1} \xrightarrow{\tau} t_n \xrightarrow{a} t'$ such that $t \mathrel{R} t_i$ for $1 \leq i \leq n$, we say that the state t_n, the action a, and the transition $t_n \xrightarrow{a} t'$ are *R-inertly reachable* from t. Note that we can use these notions in combination with (divergence-preserving) branching bisimilarity as they are all equivalence relations.

3 The Main Algorithm

In this section we give a description of the main part of the algorithm to determine the branching bisimulation equivalence classes of a given LTS $(S, Act, \rightarrow)$.

Removal of Inert Loops. As the first step of the algorithm the LTS is preprocessed to contract each τ-strongly connected component (SCC) into a single state. This step is valid since all states in a τ-SCC are branching bisimilar. For divergence-preserving branching bisimilarity, the τ-self-loops are replaced by a special non-internal action. In both cases, τ-self-loops can be suppressed and therefore, from this point onwards, as the algorithm does not change the structure of the LTS further, all τ-paths in the LTS are finite, formalised by the following lemma.

▶ **Lemma 3.1** (No τ-loops)**.** After contracting τ-SCCs into single states, there is no τ-loop in the LTS, i.e., for every sequence $s_1 \xrightarrow{\tau} s_2 \xrightarrow{\tau} \cdots \xrightarrow{\tau} s_n$ it holds that $s_i \neq s_j$ for all $1 \leq i < j \leq n$.

In every set of states $B \subseteq S$, the states without τ-transition to another state in B are called *bottom states*. The set of bottom states in B is denoted by $Bottom(B)$. Lemma 3.1 implies the following lemma:

▶ **Lemma 3.2.** For every nonempty set of states $B \subseteq S$, we have
1. $Bottom(B) \neq \emptyset$.
2. For every state $s \in B$, there is a path of τ-transitions within B leading to a bottom state.

Partition Refinement. The algorithm is a partition refinement algorithm of sets of states where two partitions of states are iteratively refined.

▶ **Definition 3.3** (Partition)**.** For a set X a partition $\mathcal{P}$ of X is a disjoint cover of X, i.e., $\mathcal{P} = \{B_i \subseteq X \mid B_i \neq \emptyset, 1 \leq i \leq k\}$ such that $B_i \cap B_j = \emptyset$ for all $1 \leq i < j \leq k$ and $X = \bigcup_{1 \leq i \leq k} B_i$.

A partition $\mathcal{Q}$ is a *refinement* of $\mathcal{P}$, and $\mathcal{P}$ is *coarser* than $\mathcal{Q}$, iff for every $B' \in \mathcal{Q}$ there is some $B \in \mathcal{P}$ such that $B' \subseteq B$.

A partition induces an equivalence relation in the following way: $s \equiv_{\mathcal{P}} t$ iff there is some $B \in \mathcal{P}$ containing both s and t. We call a transition $\mathcal{P}$-inert iff it is $\equiv_{\mathcal{P}}$-inert, i.e., exactly if it is a transition $s \xrightarrow{\tau} t$ with $s, t \in P$ for some $P \in \mathcal{P}$.

Our algorithm uses two partitions $\mathcal{B}$ and $\mathcal{C}$ of states. The main partition $\mathcal{B}$ contains *blocks,* typically denoted with the letter B, recording the current knowledge about branching bisimilarity. Two states are in different blocks iff the algorithm has found a proof that they are not branching bisimilar, formulated contrapositionally by the following invariant:

▶ **Invariant 3.4** ($\mathcal{B}$ preserves branching bisimilarity)**.** For all states $s, t \in S$, if $s \leftrightarrows_b t$, then there is some block $B \in \mathcal{B}$ such that $s, t \in B$.

Stability. We desire to make the partition $\mathcal{B}$ "stable" in the sense that blocks of $\mathcal{B}$ cannot be split further as all states in each block are branching bisimilar. This notion of stability is defined as follows. Consider two sets of states $B, B' \subseteq S$. We say that two states $s, t \in B$ are *stable under* B' iff if $s \xrightarrow{a} s'$ for any $s' \in B'$ then
- either $B = B'$ and $a = \tau$,
- or there are $t_1, \ldots, t_k \in B$ and $t' \in B'$ such that $t = t_1$, $k \geq 1$ and $t_1 \xrightarrow{\tau} \cdots \xrightarrow{\tau} t_k \xrightarrow{a} t'$.

A partition $\mathcal{B}$ of S is *stable under* a set of states $B' \subseteq S$ iff for all $B \in \mathcal{B}$ and $s, t \in B$ it holds that s and t are stable under B'. If $\mathcal{B}$ is stable under every $B \in \mathcal{B}$ we say that $\mathcal{B}$ is *stable*.

An important property is that if a partition $\mathcal{B}$ is stable, then the induced equivalence relation $\equiv_{\mathcal{B}}$ is a branching bisimulation.

Constellations. In order to remember which instabilities have already been resolved we introduce a second partition $\mathcal{C}$ of the set of states S of which $\mathcal{B}$ is a refinement. We call the blocks in $\mathcal{C}$ *constellations* and we typically denote constellations with the letter C.

The partition $\mathcal{C}$ records the current knowledge about stability by guaranteeing that $\mathcal{B}$ is always stable under each constellation $C \in \mathcal{C}$. Using Lemma 3.2 this follows from the following invariant.

▶ **Invariant 3.5** ($\mathcal{B}$ is stable under the constellations in $\mathcal{C}$)**.** If there is a transition $s \xrightarrow{a} t$ with $t \in C$ for some constellation $C \in \mathcal{C}$ and $s \in B$ for some block $B \in \mathcal{B}$, and the transition is not $\mathcal{C}$-inert (i.e., $a \neq \tau$ or $s \notin C$), then every bottom state $s' \in B$ has a transition in $s' \xrightarrow{a} C$.

The goal of the algorithm is to let the partition of constellations $\mathcal{C}$ and the partition of blocks $\mathcal{B}$ coincide. If $\mathcal{B} = \mathcal{C}$, then $\mathcal{B}$ is stable, and, using Invariants 3.4 and 3.5, the partition $\mathcal{B}$ exactly characterises branching bisimulation. As this is the core purpose of our algorithm, we formulate it as a theorem, although the proof is straightforward.

▶ **Theorem 3.6.** If $\mathcal{B} = \mathcal{C}$, then $\equiv_{\mathcal{B}} = \leftrightarrows_b$.

The main idea of the algorithm is to refine $\mathcal{C}$ until it coincides with $\mathcal{B}$. After refining $\mathcal{C}$, one may have to refine $\mathcal{B}$ to reestablish Invariant 3.5. For this to work, we use two essential subroutines, namely four-way-split$\mathcal{B}$ (Algorithm 2) and stabilise$\mathcal{B}$ (Algorithm 3). We first give a summary of what these functions do, after which we explain the overall algorithm. In Sections 4 and 5 their underlying algorithms are explained.

The function four-way-split$\mathcal{B}(B, \mathit{SmallSp}, \mathit{LargeSp})$ splits a block B in $\mathcal{B}$ in at most four sub-blocks based on two disjoint sets of transitions *SmallSp* and *LargeSp* that start in B. We require that at least one of these two sets is non-empty. If both are non-empty, every state in B can $\mathcal{B}$-inertly reach a transition in at least one of the two. The four sub-blocks are the following:

AvoidLrg: If $\mathit{LargeSp} \neq \varnothing$, AvoidLrg contains the states in B that cannot $\mathcal{B}$-inertly reach *LargeSp*. If $\mathit{LargeSp} = \varnothing$, then AvoidLrg $= \varnothing$.

AvoidSml: If $\mathit{SmallSp} \neq \varnothing$, AvoidSml contains the states in B that cannot $\mathcal{B}$-inertly reach *SmallSp*. If $\mathit{SmallSp} = \varnothing$, then AvoidSml $= \varnothing$.

ReachAlw: states in B that can always $\mathcal{B}$-inertly reach both splitters *SmallSp* and *LargeSp* if both are non-empty. Otherwise, ReachAlw consists of states in B that can always $\mathcal{B}$-inertly reach the non-empty splitter. "Always ... reach" here also means: if a $\mathcal{B}$-inert transition starts in such a state, its target state, which is in B, must also be in ReachAlw.

NewBotSt: states in B that do not fit in any of the three sets above. Original bottom states cannot end up in NewBotSt, and new bottom states will always end up in this sub-block.

Figure 1 contains two examples of how a block B' is split. Here unlabelled transitions are $\mathcal{B}$-inert transitions in B'. The next lemma expresses that new bottom states end up in NewBotSt and Invariant 3.4 is preserved under the four-way-split.

▶ **Lemma 3.7.** Consider a partition $\mathcal{B}$ and $B \in \mathcal{B}$. Assume given $a \in \mathit{Act}$ and disjoint $\mathcal{B}_{\mathrm{sml}}, \mathcal{B}_{\mathrm{lrg}} \subseteq \mathcal{B}$ with $a \neq \tau$ or $B \notin \mathcal{B}_{\mathrm{sml}} \cup \mathcal{B}_{\mathrm{lrg}}$. Let

$$\mathit{SmallSp} := B \xrightarrow{a} \bigcup \mathcal{B}_{\mathrm{sml}} \qquad \text{and} \qquad \mathit{LargeSp} := B \xrightarrow{a} \bigcup \mathcal{B}_{\mathrm{lrg}}.$$

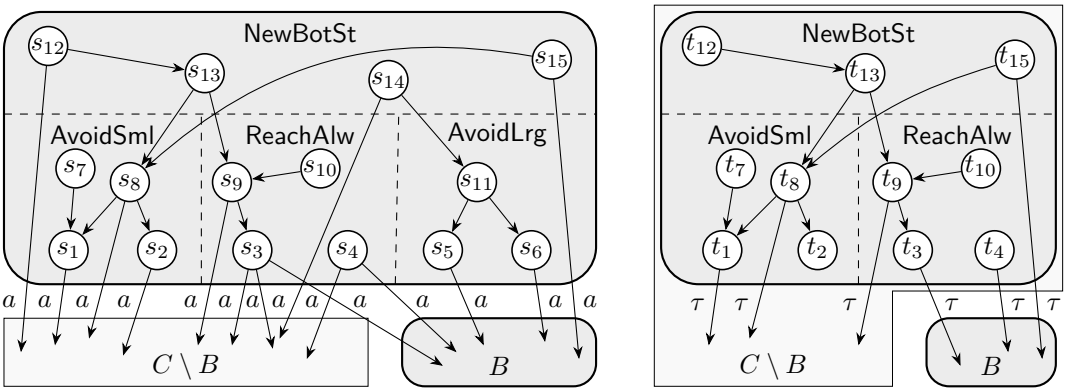


Figure 1 Left: four-way-split$\mathcal{B}(B', B' \xrightarrow{a} B, B' \xrightarrow{a} C \setminus B)$. Right: four-way-split$\mathcal{B}(B', B' \xrightarrow{\tau} B, \varnothing)$.

Assume further that $\mathit{SmallSp} \cup \mathit{LargeSp} \neq \varnothing$. Let $\mathcal{B}'$ be the partition $(\mathcal{B} \setminus \{B\}) \cup (\{\mathsf{AvoidLrg}, \mathsf{AvoidSml}, \mathsf{ReachAlw}, \mathsf{NewBotSt}\} \setminus \{\varnothing\})$ with the mentioned sets being the results of the call

four-way-split$\mathcal{B}(B, \mathit{SmallSp}, \mathit{LargeSp})$.

Then

1. For any state $s \in \mathsf{NewBotSt}$ it holds that $s \notin \mathit{Bottom}(B)$.
2. For any state $s \in \mathit{Bottom}(\mathsf{AvoidLrg}) \cup \mathit{Bottom}(\mathsf{AvoidSml}) \cup \mathit{Bottom}(\mathsf{ReachAlw})$ it holds that $s \in \mathit{Bottom}(B)$.
3. If Invariant 3.4 holds for $\mathcal{B}$, then Invariant 3.4 holds for $\mathcal{B}'$.

Proof.

1. We use contraposition. Assume that $s \in \mathit{Bottom}(B)$. Then s does not have an outgoing $\mathcal{B}$-inert transition. By construction s ends up in one of the sets AvoidSml, AvoidLrg and ReachAlw. Hence, $s \notin \mathsf{NewBotSt}$.
2. This follows by observing that if there is a bottom state $s \in B_s$ where B_s is any of the blocks AvoidLrg, AvoidSml and ReachAlw which was not a bottom state in B, then there is transition $s \xrightarrow{\tau} s'$ with $s' \in B$ but $s' \notin B_s$. Going through the various cases, it follows that s cannot be in any of the three sets B_s, and therefore must be part of NewBotSt.
3. Using that Invariant 3.4 is valid for $\mathcal{B}$, we only need to show that two states $s, t \in B$ that have been moved to different blocks in $\mathcal{B}'$ are not branching bisimilar.
   Assume $s \in \mathsf{ReachAlw}$ and $t \in \mathsf{AvoidSml}$. Then s can $\mathcal{B}$-inertly reach $\mathit{SmallSp}$. This means, using the invariant and the conditions of this lemma, it can reach a non-$\leftrightarrows_b$-inert transition in $\mathit{SmallSp}$. As $t \in \mathsf{AvoidSml}$, t cannot mimic this transition, and hence $s \not\leftrightarrows_b t$. More concretely, all states in ReachAlw and AvoidSml are not branching bisimilar. Similarly, one proves that the states in ReachAlw and AvoidLrg, respectively, the states in AvoidSml and AvoidLrg are not branching bisimilar.
   Now assume $s \in \mathsf{NewBotSt}$. We claim that s can $\mathcal{B}$-inertly reach both $\mathit{SmallSp}$ and $\mathit{LargeSp}$ unless they are empty, and s can $\mathcal{B}$-inertly reach AvoidSml or AvoidLrg. This can be seen as follows. If $s \in B$ can $\mathcal{B}$-inertly reach $\mathit{SmallSp}$ but not $\mathit{LargeSp}$ (and $\mathit{LargeSp} \neq \varnothing$), then $s \in \mathsf{AvoidLrg}$. Similarly, if $s \in B$ can $\mathcal{B}$-inertly reach $\mathit{LargeSp}$ but not $\mathit{SmallSp}$ (and $\mathit{SmallSp} \neq \varnothing$), then $s \in \mathsf{AvoidSml}$. So, s can reach all non-empty splitters. As $s \notin \mathsf{ReachAlw}$, it can $\mathcal{B}$-inertly reach a bottom state $s' \in \mathit{Bottom}(B) \setminus \mathsf{ReachAlw}$. As $s' \notin \mathsf{NewBotSt}$ (by dictum 1), we have $s' \in \mathsf{AvoidSml}$ or $s' \in \mathsf{AvoidLrg}$.
   If $t \in \mathsf{AvoidSml}$ or $t \in \mathsf{AvoidLrg}$, then s can $\mathcal{B}$-inertly reach a non-$\leftrightarrows_b$-inert transition in $\mathit{SmallSp}$ or $\mathit{LargeSp}$, respectively, that t cannot mimic.

Now consider $t \in \mathsf{ReachAlw}$. We know that s can $\mathcal{B}$-inertly reach a state $s' \in \mathsf{AvoidSml} \cup \mathsf{AvoidLrg}$, and as argued above $t \not\leftrightarrows_b s'$. Suppose we can find a state t' reachable with internal transitions from t that mimics the path from s to s' such that $t' \leftrightarrows_b s'$. Using Invariant 3.4 for $\mathcal{B}$, t' must be $\mathcal{B}$-inertly reachable and $t' \in B$. But then $t' \in \mathsf{ReachAlw}$ and $s' \not\leftrightarrows_b t'$. So, no such t' exists, and hence $s \not\leftrightarrows_b t$. ◀

The following lemma explains which calls to four-way-split$\mathcal{B}$ are needed to reestablish Invariant 3.5 after splitting a constellation $C \in \mathcal{C} \setminus \mathcal{B}$. For now we exclude blocks that contain new bottom states, i.e. sub-blocks of $\mathsf{NewBotSt}$, cf., Lemma 3.7.1 and 3.7.2.

▶ **Lemma 3.8.** Consider a partition $\mathcal{B}$ refining a partition of constellations $\mathcal{C}$ for which Invariant 3.5 holds. Let the refined set of constellations $\mathcal{C}'$ be $(\mathcal{C} \setminus \{C\}) \cup \{B, C \setminus B\}$ for some $B \in \mathcal{B}$ and $C \in \mathcal{C}$ and $B \subsetneq C$.

Let $\mathcal{B}'$ be the result of the iterative refinements $(\mathcal{B}' \setminus \{B'\}) \cup (\{\mathsf{AvoidLrg}, \mathsf{AvoidSml}, \mathsf{ReachAlw}, \mathsf{NewBotSt}\} \setminus \{\varnothing\})$ with the mentioned sets being the results of the following calls. Assume that for every action $a \in Act$ these calls take place for blocks $B' \in \mathcal{B}'$ satisfying the indicated conditions such that S is covered by these blocks B', blocks not satisfying the conditions, singleton blocks with $|B'| = 1$, or blocks that refine some block $\mathsf{NewBotSt}$:

four-way-split$\mathcal{B}(B', B' \xrightarrow{a} B, B' \xrightarrow{a} C \setminus B)$	if $a \neq \tau$ or $B' \not\subseteq C$, and there are $s_1, s_2 \in B'$ such that $s_1 \xrightarrow{a} t_1, s_2 \xrightarrow{a} t_2$ and $t_1 \in B, t_2 \in C \setminus B$,
four-way-split$\mathcal{B}(B', B' \xrightarrow{\tau} B, \varnothing)$	if $B' \subseteq C \setminus B$ and there is an $s \in B'$ such that $s \xrightarrow{\tau} t$ and $t \in B$,
four-way-split$\mathcal{B}(B', B' \xrightarrow{\tau} C \setminus B, \varnothing)$	if $B' \subseteq B$ and there is an $s \in B'$ such that $s \xrightarrow{\tau} t$ and $t \in C \setminus B$.

Then Invariant 3.5 holds for all blocks in $\mathcal{B}'$ (except for those that refine some block $\mathsf{NewBotSt}$) and $\mathcal{C}'$.

Proof. Consider a state $s \in \hat{B}$ for any block $\hat{B} \in \mathcal{B}'$ not refining any $\mathsf{NewBotSt}$. Assume that a non-$\mathcal{C}'$-inert transition $s \xrightarrow{a} t$ exists with $t \in \hat{C}$ for some constellation $\hat{C} \in \mathcal{C}'$. So, $a \neq \tau$ or $s \notin \hat{C}$. We must prove that any $s' \in \mathit{Bottom}(\hat{B})$ has a transition $s' \xrightarrow{a} t'$ with $t' \in \hat{C}$.

Let $\hat{B}'$ be the block $\hat{B} \subseteq \hat{B}' \in \mathcal{B}$. Note that $s' \in \mathit{Bottom}(\hat{B}')$ by Lemma 3.7.2.

Case $\hat{C} \not\subseteq C$: That means that $\hat{C} \in \mathcal{C}$. Using Invariant 3.5 (for $\hat{B}' \in \mathcal{B}$ and $\hat{C} \in \mathcal{C}$), s' has a transition $s' \xrightarrow{a} t'$ with $t' \in \hat{C}$.

Case $\hat{B} \subseteq C$, $a = \tau$ and $\hat{C} = B$: If $\hat{B} \subseteq B = \hat{B}'$, then $s \xrightarrow{\tau} t$ were $\mathcal{C}'$-inert. So, we can assume $\hat{B} \subseteq C \setminus B$. There is a block $\hat{B}''$ with $\hat{B} \subseteq \hat{B}'' \subseteq \hat{B}'$ such that a call

four-way-split$\mathcal{B}(\hat{B}'', \hat{B}'' \xrightarrow{\tau} B, \varnothing)$

must have taken place. In this case $\hat{B}$ is a sub-block of $\mathsf{ReachAlw}$ or $\mathsf{NewBotSt}$. In the last case there is nothing to check. But $\hat{B} \subseteq \mathsf{ReachAlw}$ implies that $s' \in \mathit{Bottom}(\hat{B})$ has a transition $s' \xrightarrow{\tau} t'$ with $t' \in B$, as had to be shown.

Case ($\hat{B} \not\subseteq C$ or $a \neq \tau$) and $\hat{C} = B$: If $(\hat{B} \xrightarrow{a} C \setminus B) = \varnothing$, we know by Invariant 3.5 (for $\hat{B}' \in \mathcal{B}$ and $C \in \mathcal{C}$) that $s' \in \mathit{Bottom}(\hat{B})$ has a transition $s' \xrightarrow{a} t'$ with $t' \in C$, and as the transition cannot go to $C \setminus B$, it must go to B, which we had to prove.

Otherwise, $(\hat{B} \xrightarrow{a} C \setminus B) \neq \varnothing$, and there is a block $\hat{B}''$ with $\hat{B} \subseteq \hat{B}'' \subseteq \hat{B}'$ for which a call

four-way-split$\mathcal{B}(\hat{B}'', \hat{B}'' \xrightarrow{a} B, \hat{B}'' \xrightarrow{a} C \setminus B)$

took place with $\hat{B}$ a sub-block of $\mathsf{AvoidLrg}$, $\mathsf{ReachAlw}$ or $\mathsf{NewBotSt}$. The last case does not lead to a proof obligation. But $\hat{B} \subseteq \mathsf{AvoidLrg}$ or $\hat{B} \subseteq \mathsf{ReachAlw}$ implies that $s' \in \mathit{Bottom}(\hat{B})$ has a transition $s' \xrightarrow{a} t'$ with $t' \in B$, as had to be shown.

Algorithm 1 Branching bisimulation partitioning.

1.1:	Find τ-SCCs and contract each of them to a single state	$O(\lvert Act\rvert + m)$
1.2:	$\mathcal{C} := \{S\}$; $\mathcal{B} := \{S\}$; $has_new_bottom_states(S) := true$	
1.3:	Call stabilise$\mathcal{B}$() to create the initial partition	
1.4:	**while** a $B \in \mathcal{B}$ exists such that $B \notin \mathcal{C}$ **do**	$\leq n$ iterations
1.5:	Select some $B \in \mathcal{B}$ and $C \in \mathcal{C}$ such that $B \subseteq C$ and $\lvert B\rvert \leq \frac{1}{2}\lvert C\rvert$	
1.6:	$\mathcal{C} := (\mathcal{C} \setminus \{C\}) \cup \{B, C \setminus B\}$ and maintain data structures accordingly	
1.7:	$\mathcal{M} := \{s \xrightarrow{a} t \mid s \in S, a \in Act, t \in B,$ and $a \neq \tau$ or $s \notin B\}$	$O(\lvert in(B)\rvert)$
1.8:	Split B under the outgoing transitions that have just become non-$\mathcal{C}$-inert by calling four-way-split$\mathcal{B}(B, B \xrightarrow{\tau} (C \setminus B), \varnothing)$	+ split
1.9:	**while** $\mathcal{M} \neq \emptyset$ **do**	$\leq \lvert in(B)\rvert$ iter'ns
1.10:	Pick some BLC set $SmallSp := (B' \xrightarrow{a} B) \subseteq \mathcal{M}$ for some block B'	
1.11:	$\mathcal{M} := \mathcal{M} \setminus SmallSp$	
1.12:	**if** $\lvert B'\rvert > 1 \wedge \neg has_new_bottom_states(B')$ **then**	
1.13:	**if** $B' \subseteq C \wedge a = \tau$ **then**	
1.14:	Split B' under the transitions that have become non-$\mathcal{C}$-inert by calling four-way-split$\mathcal{B}(B', SmallSp, \varnothing)$	$O(1)$ + split
1.15:	**else**	
1.16:	$LargeSp := B' \xrightarrow{a} (C \setminus B)$	
1.17:	**if** $LargeSp \neq \varnothing$ **then**	
1.18:	Split B' by calling four-way-split$\mathcal{B}(B', SmallSp, LargeSp)$	
1.19:	Call stabilise$\mathcal{B}$() to stabilise blocks with new bottom states	
1.20:	**return** $\mathcal{B}$	

Case $\hat{B} \subseteq C$, $a = \tau$ and $\hat{C} = C \setminus B$: If $\hat{B} \subseteq C \setminus B$, then $s \xrightarrow{\tau} t$ were $\mathcal{C}'$-inert. So we can assume $\hat{B} \subseteq B = \hat{B}'$. There is a block $\hat{B}''$ with $\hat{B} \subseteq \hat{B}'' \subseteq B$ such that a call

four-way-split$\mathcal{B}(\hat{B}'', \hat{B}'' \xrightarrow{\tau} C \setminus B, \varnothing)$

must have taken place, where $\hat{B}$ is a sub-block of ReachAlw or NewBotSt. The latter case is not interesting. Again, $\hat{B} \subseteq$ ReachAlw implies that $s' \in Bottom(\hat{B})$ has a transition $s' \xrightarrow{\tau} t'$ with $t' \in C \setminus B$, as had to be shown.

Case ($\hat{B} \not\subseteq C$ or $a = \tau$) and $\hat{C} = C \setminus B$: If $(\hat{B} \xrightarrow{a} B) = \varnothing$, we know by Invariant 3.5 (for $\hat{B}' \in \mathcal{B}$ and $C \in \mathcal{C}$) that $s' \in Bottom(\hat{B})$ has a transition $s' \xrightarrow{a} t'$ with $t' \in C$, and as the transition cannot go to B, it must go to $C \setminus B$, which we had to prove.
Otherwise, $(\hat{B} \xrightarrow{a} B) \neq \varnothing$, and there is a block $\hat{B}''$ such that $\hat{B} \subseteq \hat{B}'' \subseteq \hat{B}'$ and a call

four-way-split$\mathcal{B}(\hat{B}'', \hat{B}'' \xrightarrow{a} B, \hat{B}'' \xrightarrow{a} C \setminus B)$

must have taken place, and $\hat{B}$ is a subset of AvoidSml or ReachAlw or NewBotSt. We can ignore the latter case. But $\hat{B} \subseteq$ AvoidSml or $\hat{B} \subseteq$ ReachAlw implies that $s' \in Bottom(\hat{B})$ has a transition $s' \xrightarrow{a} t'$ with $t' \in C \setminus B$, as had to be shown. ◀

Blocks with new bottom states, which were excluded by Lemma 3.8, are handled by the function stabilise$\mathcal{B}$. It assumes that blocks without new bottom states are already stable and stabilises the blocks with new bottom states, which concretely means that it refines them under all their outgoing non-$\mathcal{C}$-inert transitions such that for the resulting partition Invariant 3.5 holds, while maintaining Invariant 3.4.

Main Algorithm. We are now in the situation to explain Algorithm 1. First we need to take care that the Invariants 3.4 and 3.5 become valid. Therefore, in line 1.2 we set the partition $\mathcal{B}$ and the set of constellations $\mathcal{C}$ to $\{S\}$. This makes Invariant 3.4 trivially true.

If a block B has new bottom states, we use the predicate $has_new_bottom_states(B)$ to indicate so. This is not only the case when block B is a refinement of some block NewBotSt but it also applies to the initial block, and this is explicitly indicated in line 1.2. Invariant 3.5 is now made valid by one call to stabilise$\mathcal{B}$ in line 1.3. Concretely, this call splits $\{S\}$ into a partition $\mathcal{B}$ such that all bottom states in each block B in $\mathcal{B}$ have exactly the same outgoing non-$\mathcal{B}$-inert actions, while keeping branching bisimilar states in the same block.

As stated above, the purpose of the main algorithm is to refine $\mathcal{C}$ such that it ultimately coincides with $\mathcal{B}$. This happens in the loop starting at line 1.4. If $\mathcal{B}$ and $\mathcal{C}$ do not coincide, there is at least one block $B \subseteq C$ with size smaller than or equal to $\frac{1}{2}|C|$. We replace the constellation C in $\mathcal{C}$ by two new constellations B and $C \setminus B$ (line 1.6).

Our main task is to make Invariant 3.5 valid again. This is done by taking care that all calls to four-way-split$\mathcal{B}$ as required by Lemma 3.8 are performed. This guarantees that Invariant 3.5 is valid except for blocks of the form NewBotSt. Therefore, one extra call is needed to stabilise$\mathcal{B}$ in line 1.19.

In order to effectively carry out the four-way-splits, transitions are organised in *block-label-constellation sets (BLC sets)*. Every BLC set contains the transitions $B \xrightarrow{a} C$ for a specific block $B \in \mathcal{B}$, action $a \in Act$, and constellation $C \in \mathcal{C}$. Whenever a constellation or a block is split (lines 1.6 and 2.45) these BLC sets are updated.

In line 1.8 there is a call to four-way-split$\mathcal{B}(B, B \xrightarrow{\tau} (C \setminus B), \varnothing)$ that corresponds with the third case in Lemma 3.8. So, that case has been covered.

For the remainder we put all non-$\mathcal{C}$-inert transitions to block B in a set $\mathcal{M}$. By traversing the BLC sets in $\mathcal{M}$ the other required calls to four-way split are done. In line 1.5 such a BLC set $SmallSp = B' \xrightarrow{a} B$ for some block B' is selected. For the complexity, it is important that $|B| < \frac{1}{2}|C|$ as this implies that the selected BLC set is small, hence its name.

If B' has only one state or is a refinement of some block NewBotSt and hence has new bottom states, according to Lemma 3.8 it does not need to be considered to be split (line 1.12).

Otherwise, there are two situations. The first one is when $a = \tau$ and B' was part of the constellation C (line 1.13). As $\mathcal{M}$ does not contain τ-transitions from B to B, this means $B' \in C \setminus B$ and the required second call four-way-split$\mathcal{B}(B', SmallSp, \varnothing)$ in Lemma 3.8 is made. The other situation, i.e., the **else** part at lines 1.15–1.18, exactly satisfies the conditions of the first mentioned call to four-way-split in Lemma 3.8. As the algorithm traverses all BLC sets that are part of $\mathcal{M}$, the whole set S is covered as required in Lemma 3.8.

From Lemma 3.8 it follows that at line 1.19 of the algorithm, Invariant 3.5 holds for all blocks, except for those that are of the shape NewBotSt. As stated above one call to stabilise$\mathcal{B}$ is sufficient to restore Invariant 3.5 for all blocks. Just for the record, Invariant 3.4 remains valid, as each call to four-way-split$\mathcal{B}$ preserves it (see Lemma 3.7.3), and, as explained in Section 5, stabilise$\mathcal{B}$ refines $\mathcal{B}$ only using four-way-split$\mathcal{B}$ also.

This means that at line 1.4 of Algorithm 1 both invariants hold. If the condition of the **while** is not valid, the partitions $\mathcal{B}$ and $\mathcal{C}$ are equal, and as argued above, these resulting partitions coincide exactly with branching bisimulation. As the initial set of states S is finite, the partition $\mathcal{C}$ can only be refined finitely often, which happens in the body of the while loop. This means that this algorithm always terminates with the required answer.

4 Four-Way Split

This section explains how four-way-split$\mathcal{B}(B, SmallSp, LargeSp)$ refines block B under two splitters, i.e., sets of transitions *SmallSp* and *LargeSp*. The sets contain all transitions from B with a given label a to one or two given constellations.

Algorithm 2 Refine a block under one or two sets of transitions.

2.1: **function** four-way-split$_{\mathcal{B}}$(block B, small splitter $SmallSp$, large splitter $LargeSp$)
2.2: ReachAlw := $\{s \in Bottom(B) \mid (SmallSp = \varnothing \vee \exists s \xrightarrow{a} t \in SmallSp) \wedge (LargeSp = \varnothing \vee \exists s \xrightarrow{a} t \in LargeSp)\}$
2.3: AvoidLrg := $\{s \in Bottom(B) \mid (SmallSp = \varnothing \vee \exists s \xrightarrow{a} t \in SmallSp) \wedge \neg(LargeSp = \varnothing \vee \exists s \xrightarrow{a} t \in LargeSp)\}$
2.4: AvoidSml := $Bottom(B) \setminus$ (ReachAlw $\cup$ AvoidLrg)
2.5: **if** AvoidLrg $= \varnothing \wedge$ AvoidSml $= \varnothing$ **then**
2.6: **return** // B is stable under $SmallSp$ and $LargeSp$.
2.7: pot-ReachAlw := $\{s \in B \setminus Bottom(B) \mid \exists s \xrightarrow{a} t \in SmallSp \wedge (LargeSp = \varnothing \vee \exists s \xrightarrow{a} t \in LargeSp)\}$
2.8: HitSmall := $\{s \in B \setminus Bottom(B) \mid \exists s \xrightarrow{a} t \in SmallSp \wedge \neg(LargeSp = \varnothing \vee \exists s \xrightarrow{a} t \in LargeSp)\}$
2.9: **for all** $s \in$ pot-ReachAlw **do**
2.10: $s.count := |s \xrightarrow{\tau} B|$
2.11: NewBotSt := $\varnothing$; pot-AvoidLrg := $\varnothing$; pot-AvoidSml := $\varnothing$

$O(|SmallSp|)$ or $O(|Bottom(B) \xrightarrow{a} S \cap LargeSp|)$

2.12: **begin coroutines**
2.13: If some subset becomes too large at any time, abort its coroutine immediately.

cor $R =$ ReachAlw, AvoidSml, AvoidLrg **do**
2.14: **for all** $s \in R$ **do**
2.15: **for all** $t \xrightarrow{\tau} s \in in(s)$ **do**
2.16: **if** $t \notin B \setminus$ NewBotSt **then**
2.17: Skip $t \xrightarrow{\tau} s$ (go to line 2.16)
2.18: **if** $t \notin$ pot-R **then**
2.19: **if** $R =$ AvoidSml $\wedge\, t \in$ HitSmall $\vee\, \exists R'.t \in$ pot-R' **then**
2.20: Add t to NewBotSt
2.21: Skip $t \xrightarrow{\tau} s$ (go to line 2.16)
2.22: Add t to pot-R
2.23: $t.count := |t \xrightarrow{\tau} B|$
2.24: $t.count := t.count - 1$
2.25: **if** $t.count \neq 0$ **then**
2.26: Skip $t \xrightarrow{\tau} s$ (go to line 2.16)
2.27: **if** $R =$ AvoidLrg **then**
2.28: **for all** $t \xrightarrow{b} u \in out(t)$ **do**
2.29: **if** $t \xrightarrow{b} u \in LargeSp$ **then**
2.30: Add t to NewBotSt
2.31: Skip $t \xrightarrow{\tau} s$ (go to line 2.16)
2.32: Add t to R
2.33: Declare R finished
2.34: **if** 3 coroutines have finished **then**
2.35: Terminate the coroutines
2.36: NewBotSt := NewBotSt $\cup$ (pot-$R \setminus R$)
2.37: **if** ReachAlw&AvoidLrg are finished **then**
2.38: NewBotSt := NewBotSt $\cup$ HitSmall $\setminus$ (ReachAlw $\cup$ AvoidLrg)

$O(|in(R)|)$ or $O(|out(\text{AvoidLrg})| + |out(Bottom(\text{NewBotSt}))|)$ $O(|in(R)|)$ $O(|SmallSp|)$

loop forever
for all unexplored $s \in$ NewBotSt **do**
for all $t \xrightarrow{\tau} s \in in(s)$ **do**
if $t \in B$ **then**
Add t to NewBotSt
if 2 coroutines have finished **then**
if $LargeSp$ is fully explored $\vee$ AvoidLrg is finished **then**
Declare NewBotSt finished
Terminate the coroutines
Pick unexplored $t \xrightarrow{a} s \in LargeSp$
if $t \notin$ ReachAlw $\cup$ AvoidSml $\wedge\, t \notin Bottom(B)$ **then**
Add t to NewBotSt

$O(|in(R)|)$ or $O(|in(\text{NewBotSt})| + |out(\text{NewBotSt})| + |out(finished)|)$

2.39: **end coroutines**
2.40: **if** NewBotSt is unfinished $\wedge$ ReachAlw $\cup$ AvoidLrg $\cup$ AvoidSml $= B$ **then**
2.41: Declare NewBotSt finished and the largest subset aborted
2.42: Create sub-blocks for the finished, non-empty subsets
2.43: Update transitions that have become non-$\mathcal{B}$-inert and BLC sets
2.44: **if** NewBotSt $\neq \varnothing$ **then**
2.45: *has_new_bottom_states*(NewBotSt) := *true*
2.46: **return**

$O(|in(finished)| + |out(finished)|)$

Splitting block B must be done in a time proportional to the size of *SmallSp* and the sizes of all but the largest block into which B is split. The principle "process the smaller half" then implies that every state and transition is only visited $O(\log n)$ times. However, we do not know a priori which of the three parts are the smallest. Therefore, we start four coroutines for the four parts, run them in lockstep and abort the coroutine that belongs to the largest part. This ensures that time proportional to the three smallest parts is used.

Preconditions. At least one splitter must be non-empty. If both splitters are given (as in line 1.18 of Algorithm 1), it is known that every bottom state of B has a transition in at least one of the two splitters. We use data structures such that it is possible to quickly determine whether states with a transition in *SmallSp* also have a transition in *LargeSp*.

We now go through the pseudocode shown as Algorithm 2.

Preparation: Splitting the Bottom States. In lines 2.2–2.11, the algorithm initialises a number of sets of states, mainly to decide for every bottom state to which part it belongs.

In lines 2.2–2.4 in case $\mathit{SmallSp} \neq \varnothing$ every transition in *SmallSp* is visited, and its source state is moved to ReachAlw or AvoidLrg, depending on whether it has a transition in *LargeSp* or not. Bottom states that have not been moved belong to AvoidSml. In case $\mathit{SmallSp} = \varnothing$, the transitions in *LargeSp* that start in a bottom state are traversed, to move these bottom states to ReachAlw. The states that have not been moved belong to AvoidLrg.

If some non-bottom state has a transition in *SmallSp*, it is not immediately known to which part it belongs; it might happen that it has a $\mathcal{B}$-inert transition to another part. If the non-bottom state has a transition in the small splitter but not in $\mathit{LargeSp} \neq \varnothing$ (like s_{15} in Figure 1 at the left), it may end up in any part except AvoidSml. To register this such a state is temporarily added to HitSmall in line 2.8. It will be processed further in line 2.20 left.

If the non-bottom state has a transition in all (non-empty) splitters, it is potentially in ReachAlw, unless it has a $\mathcal{B}$-inert transition to one of the other sub-blocks. In the latter case, it will be in NewBotSt. We add it temporarily to pot-ReachAlw to register this in line 2.7.

If all bottom states belong to ReachAlw, the split is trivial (lines 2.5–2.6). Even if all bottom states belong to one of AvoidLrg or AvoidSml, there still can be some non-bottom states with transitions causing a split. Such states will be moved to NewBotSt.

Coroutines: Extending the Splits to Non-bottom States. After all bottom states have been distributed over the initial sets, we have to extend the distribution to the non-bottom states. To stay within the time bounds, we need to find all the states in the three smallest parts in time proportional to the number of their incoming and outgoing transitions.

Of the four coroutines, three (for ReachAlw, AvoidSml and AvoidLrg) are almost the same. Their code is shown on the left of lines 2.12–2.41, where we use the keyword **cor** to indicate the three coroutines. The basic idea is the following. If all outgoing $\mathcal{B}$-inert transitions of some non-bottom state go to the same set, the state also belongs to this set. To avoid checking any transition more than once, we go through all incoming $\mathcal{B}$-inert transitions of the respective set and count how many outgoing $\mathcal{B}$-inert transitions of its source are not yet known to point to this set. If the $\mathcal{B}$-inert transitions point to different sets, the state belongs to NewBotSt (line 2.21 left).

If the number of unknown transitions is 0, the source is added to the respective set, unless it has an incompatible transition in a splitter: states with a transition in both splitters can only be in ReachAlw or NewBotSt and were therefore already added to pot-ReachAlw; this ensures that they move to NewBotSt as soon as it is found out that they have a transition to

Algorithm 3 Stabilise the partition with respect to new bottom states.

3.1: **function** stabilise$\mathcal{B}$()
3.2: $\hat{\mathcal{Q}} := \varnothing$
3.3: **for all** blocks B with $has_new_bottom_states(B)$ **do**
3.4: $\quad \hat{\mathcal{Q}} := \hat{\mathcal{Q}} \cup \{s \xrightarrow{a} t \mid s \in B \wedge (a \neq \tau \vee constellation(s) \neq constellation(t))\}$ } $O(|out(Bottom^+(B))|)$
3.5: $\quad$ **for all** $s \in Bottom(B)$ **do**
3.6: $\quad\quad$ Move the non-$\mathcal{C}$-inert transitions out of s to the front of their BLC sets } $O(|out(Bottom(B))|)$
3.7: $\quad has_new_bottom_states(B) := false$
3.8: **while** $\hat{\mathcal{Q}} \neq \varnothing$ **do** } $\leq |out(Bottom^+(B))|$ iter'ns
3.9: $\quad$ Pick some BLC set $Sp := (B \xrightarrow{a} C) \subseteq \hat{\mathcal{Q}}$
3.10: $\quad \hat{\mathcal{Q}} := \hat{\mathcal{Q}} \setminus Sp$
3.11: $\quad$ **if** $|B| > 1$ **then** } $O(1)$ + split
3.12: $\quad\quad$ Split B under Sp by calling four-way-split$\mathcal{B}(B, \varnothing, Sp)$
3.13: $\quad\quad$ **if** the split produced some NewBotSt $\neq \varnothing$ **then**
3.14: $\quad\quad\quad \hat{\mathcal{Q}} := \hat{\mathcal{Q}} \cup \{s \xrightarrow{a} t \mid (s \xrightarrow{a} t) \notin \hat{\mathcal{Q}} \wedge s \in \text{NewBotSt} \wedge (a \neq \tau \vee constellation(s) \neq constellation(t))\}$ } $O\left(|out(Bottom^+(\text{NewBotSt})) \setminus \hat{\mathcal{Q}}|\right)$
3.15: $\quad\quad\quad$ **for all** $s \in Bottom(\text{NewBotSt})$ **do**
3.16: $\quad\quad\quad\quad$ Move the non-$\mathcal{C}$-inert transitions out of s to the front of their BLC sets } $O\left(|out(Bottom(\text{NewBotSt}))|\right)$
3.17: $\quad\quad\quad has_new_bottom_states(\text{NewBotSt}) := false$
3.18: **return**

AvoidSml or AvoidLrg. – States with a transition in *SmallSp* but not in *LargeSp* cannot be in AvoidSml. If they would be added to that set, the test in line 2.20 left ensures that they will be added to NewBotSt instead. – States with a transition in *LargeSp* but not in *SmallSp* cannot be in AvoidLrg. But as the algorithm is not allowed to go through all transitions in *LargeSp* a priori, they are more difficult to handle. Only if the number of unknown transitions is 0 and the state is still a candidate for AvoidLrg, we are allowed to spend the time to check whether it has a transition in *LargeSp* by looking through its non-$\mathcal{C}$-inert outgoing transitions in lines 2.29–2.33 left. Section 6 explains why this fits the time bounds.

When all incoming $\mathcal{B}$-inert transitions of a set have been visited, no more states can be added, so that set is finished. The states that had some $\mathcal{B}$-inert transitions to the set but still have unknown transitions must have transitions to some other set as well, so they are added to NewBotSt in line 2.38 left. If AvoidSml is the only coroutine that still runs on the left side, any unassigned states remaining in HitSmall also belong to NewBotSt, as they cannot be in AvoidSml (see lines 2.39–2.40 left).

The fourth coroutine, the one for NewBotSt, follows a slightly simpler principle. Every $\mathcal{B}$-inert predecessor of a NewBotSt-state is also in NewBotSt, independent of other transitions it might have. This happens in line 2.18 right. The other three coroutines may also occasionally add states to NewBotSt and therefore, different from the first three coroutines, we cannot end the coroutine as soon as all incoming $\mathcal{B}$-inert transitions of the current NewBotSt have been visited. Instead, we wait in an (outer) infinite loop. The outer loop can terminate when at least AvoidLrg and one other coroutine have finished.

If only the coroutines for AvoidLrg and NewBotSt are still running, the latter will go through *LargeSp* to find states that cannot be in AvoidLrg in lines 2.24–2.27 right. If the source state of some transition in *LargeSp* has not yet been inserted into a sub-block, it cannot be in AvoidLrg, so it must be in NewBotSt.

5 Stabilising New Bottom States

In this section we explain stabilise$\mathcal{B}$ given in Algorithm 3. The procedure stabilise$\mathcal{B}$ goes through all blocks with new bottom states and completely stabilises them.

Algorithm 3 shows the pseudocode of the stabilisation. In lines 3.3–3.7, all transitions out of the states in blocks with new bottom states are collected in $\hat{\mathcal{Q}}$. Additionally, all non-$\mathcal{C}$-inert transitions out of bottom states, which are all new, are put to the front of their BLC sets. In line 3.12 there is a call to four-way-split$\mathcal{B}$ that only traverses these transitions in front of the BLC set to determine whether B needs to be split.

In lines 3.8–3.12, each BLC set is iteratively chosen as potential splitter Sp and its source block B is stabilised under it. In the preparation phase of four-way-split$\mathcal{B}$, we can now visit the transitions in Sp starting in bottom states without spending time to pick them from between those starting in non-bottom states. If this refinement leads to further new bottom states, the sub-block NewBotSt is not empty. This block with new bottom states is immediately added to $\hat{\mathcal{Q}}$ as it requires to be stabilised again, see lines 3.13–3.17.

To ensure that every transition out of a new bottom state is visited only a bounded number of times, we have to take into account that some transitions out of the set NewBotSt may already be in $\hat{\mathcal{Q}}$ and do not need to be added a second time.

6 Complexity

We attribute computation steps to certain transitions or states in the LTS to account for the time bound. There are three kinds of timing attributions:

C. When a constellation C is split into B and $C \setminus B$, time is spent proportionally to the states and incoming and outgoing transitions of B.

B. When a block B is split into AvoidLrg, AvoidSml, ReachAlw and NewBotSt, time is spent proportionally to the states and incoming and outgoing transitions of the three smallest sub-blocks.

N. When a new bottom state is found, time is spent proportionally to this state and its outgoing transitions.

We first formulate three lemmas to distribute all execution steps (except the initialisation) over the three cases above, and then state the main timing result.

▶ **Lemma 6.1.** Every step in four-way-split$\mathcal{B}$ (Algorithm 2) falls under one of the timing items C, B, or N above.

Proof. We can ignore the coroutine for the largest sub-block because it is aborted as soon as it is clearly the largest (line 2.13) or the three other coroutines are finished. The time it spends is at most proportional to the time attribution of the second-largest coroutine.

Initialisation (lines 2.2–2.11): If the function is called from the main algorithm, *SmallSp* either consists of outgoing transitions of B (line 1.8) or of incoming transitions of B (lines 1.14 and 1.18). Then the sets ReachAlw, AvoidLrg, AvoidSml, pot-ReachAlw and HitSmall can be initialised by going through the transitions in *SmallSp*, which falls under timing item C. Note that we require that for states with a transition in *SmallSp*, one can determine in time $O(1)$ whether they have a transition in *LargeSp*. This allows to distinguish ReachAlw from AvoidLrg and pot-ReachAlw from HitSmall.

If the function is called from stabilise$\mathcal{B}$ (line 3.12), $SmallSp = \varnothing$, and *LargeSp* consists of outgoing transitions of a block with new bottom states. Then the sets ReachAlw and AvoidLrg can be distinguished by going through those transitions in *LargeSp* that start in

(new) bottom states, which falls under timing item N. The other sets in the initialisation are empty. Note that we only spend time on transitions in *LargeSp* starting in *bottom* states because we singled them out in lines 3.6 and 3.16.

All coroutines: Every coroutine generally runs a loop over all states in the respective sub-block and its incoming transitions. This can immediately be attributed to timing item B.

Coroutine for AvoidLrg (lines 2.29–2.33 left): The coroutine for AvoidLrg, once it has found a state t with $t.count = 0$, still needs to check whether t has a transition in *LargeSp* that could not be handled during initialisation. Depending on the outcome of this check, we attribute the time spent on the loop in lines 2.29–2.33 left differently:

- If t has no transition in *LargeSp*, we conclude that $t \in$ AvoidLrg, and we can attribute the time to the outgoing transitions of AvoidLrg, i.e. to timing item B.
- If, however, t has a transition in *LargeSp*, it is added to NewBotSt in line 2.32 left, and it becomes a new bottom state: all its outgoing $\mathcal{B}$-inert transitions point to another sub-block (namely AvoidLrg). Therefore we can attribute this to timing item N.

Finalising a sub-block $R =$ ReachAlw, AvoidSml or AvoidLrg: In line 2.38 left, states in pot-$R \setminus R$ are moved to NewBotSt. Every such state has a transition to R, so this is attributed to timing item B. In lines 2.39–2.40 left, the HitSmall-states are moved to NewBotSt. Because $|\text{HitSmall}| \leq |\mathit{SmallSp}|$, we can attribute this to timing item C.

Coroutine for NewBotSt: This coroutine has an outer infinite loop in line 2.14 right, when all states that are known to be in NewBotSt are already explored and the coroutine has to wait until a different coroutine adds another state to NewBotSt. Note that this will only happen as long as at least two coroutines different from NewBotSt are running. So, we can attribute the waiting time to the smaller of the two sub-blocks. When only the coroutines for AvoidLrg and NewBotSt are unfinished, the latter goes through the transitions in *LargeSp* in lines 2.24–2.27 right. Source states of these transitions cannot be in AvoidLrg, so they either are in a finished sub-block (and then that sub-block is not the largest one, so we are allowed to spend time on its outgoing transitions under timing item B) or they are in NewBotSt (and then they also fall under timing item B).

Final administration (lines 2.42–2.46): This can be organised in a way that only the states in the three smallest sub-blocks and their outgoing transitions are moved to new blocks and new BLC sets, respectively. So, we attribute it to timing item B. ◀

▶ **Lemma 6.2.** Every step in stabilise$\mathcal{B}$ (Algorithm 3) falls under one of the timing items C, B, or N above.

Proof. It can happen that we have to stabilise a block with new bottom states under BLC sets whose transitions all start in non-bottom states, but then *after* the respective call to four-way-split$\mathcal{B}$, this BLC set will contain a transition that starts in a new bottom state. We denote the set of the new bottom states together with these prospective new bottom states by "$\mathit{Bottom}^+(B)$".

Initialisation (lines 3.3–3.7): If one adds transitions to $\hat{\mathcal{Q}}$ in constant time per BLC set in line 3.4, then time $O(|\mathit{Bottom}^+(B)|)$ is spent. Lines 3.5–3.7 access new bottom states and their outgoing transitions. All these steps can be attributed to timing item N.

Main loop (lines 3.8–3.12): In every iteration one BLC set in $\hat{\mathcal{Q}}$ is handled, so there cannot be more iterations than transitions starting in $\mathit{Bottom}^+(B)$ (summed over all blocks B with new bottom states). The call to four-way-split$\mathcal{B}$ is attributed according to Lemma 6.1.

Adding another new bottom block (lines 3.13–3.17): This is the same operation as the loop body for the initialisation (lines 3.4–3.7), so we can attribute the timing in the same way. When adding new transitions to $\hat{\mathcal{Q}}$, one only has to take care not to spend time on BLC sets that have been added already earlier. ◀

▶ **Lemma 6.3.** Every step in the main loop in Algorithm 1 (lines 1.4–1.19) falls under one of the timing items C, B, or N above.

Proof. Splitting a constellation (lines 1.4–1.8): This part of the algorithm handles the states and transitions of B and is attributed to timing item C.

Stabilising under B and $C \setminus B$ (lines 1.9–1.19): Except for the calls to four-way-split$\mathcal{B}$ and stabilise$\mathcal{B}$ (which fall under Lemmas 6.1 and 6.2), these lines use constant time per iteration. Because $\mathcal{M} \subseteq in(B)$, there are not too many iterations. ◀

▶ **Theorem 6.4.** Under the reasonable assumption $|Act| \in O(m)$ and $n \in O(m)$, the running time of Algorithm 1 including its subroutines is in $O(m \log n)$.

Proof. Initialisation. The calculation of the τ-SCCs (line 1.1) takes $O(m)$ time via a standard algorithm [13]. Line 1.2 includes constructing the initial BLC sets. The essential step is to (bucket-)sort the transitions according to actions in time $O(|Act| + m) = O(m)$.

Steps under timing item C. State s can be in the small new constellation B (line 1.6) at most $\log_2 n$ times. Whenever this happens, time in $O(1 + |in(s)| + |out(s)|)$ is spent. Summing over all states, the total running time under timing item C is in $O(m \log n)$.

Steps under timing item B. State s can be in a small sub-block (when calling four-way-split$\mathcal{B}$) at most $\log_2 n$ times. Whenever this happens, time in $O(1 + |in(s)| + |out(s)|)$ is spent. Summing over all states, the total running time under timing item B is in $O(m \log n)$.

Steps under timing item N. State s is treated as new bottom state at most once during the whole algorithm. When this happens, time in $O(1 + |out(s)|)$ is spent. Summing over all states, the total running time under timing item N is in $O(m)$.

As now every part of the algorithm is accounted for, either directly or through one of the above lemmas, the result follows. ◀

Data Structures that Satisfy the Timing Constraints. We believe that many data structures are straightforward to implement, and we only describe the less obvious data structures here.

Transitions are stored in three orderings: incoming transitions per state (used e.g. in lines 1.6–1.7 and 2.16), outgoing transitions per state (used e.g. in lines 2.30 left and 3.6), and transitions per BLC set (used e.g. in lines 2.24 right and 3.4). The outgoing transitions are grouped per label and target constellation, and every block has a list of BLC sets with transitions that start in this block.

When splitting a constellation (line 1.6), the groups for outgoing transitions per state to C are split into two, such that the groups for *SmallSp* and *LargeSp* are next to each other. This allows to find out quickly whether a state with an outgoing transition in one of them also has a transition in the other (used to distinguish ReachAlw from AvoidLrg and pot-ReachAlw from HitSmall in lines 2.2–2.8). Similarly, the list entries for BLC sets containing transitions to C are split into two adjacent ones, and line 1.16 can find *LargeSp* (given *SmallSp*) in time $O(1)$.

During stabilise$\mathcal{B}$, we place BLC sets that are in $\hat{\mathcal{Q}}$ near the end of the list, so that line 3.14 only spends time on BLC sets that are currently not in $\hat{\mathcal{Q}}$.

To be absolutely sure that our correctness and timing analysis, as well as the implementation are correct we instrumented the implementation with both assertions and counters counting each operation on states and transitions and ran it on a large number of transition systems, most of which were randomly generated. Besides the invariants and the overall time bounds, it is checked that after the coroutines are finished, the counters for the largest sub-block have not been increased more often than the others, and also that the coroutine for NewBotSt did not wait too often without doing assignable work.

Table 1 Branching bisimulation benchmarks. – Running times are rounded to significant digits; trailing zeroes without following decimal point are insignificant. Bold times indicate the algorithm that is faster with 95% confidence.

	After SCC		After reduction		JGKW		This paper	
	#states	#trans	#states	#trans	time	memory	time	memory
vasy_574_13561	.57M	1.4M	3,577	16,168	20 s	1.3 G	**10 s**	**.91G**
vasy_6020_19353	1.4 M	3.9M	256	510	**1.7 s**	.83G	3. s	.83G
vasy_1112_5290	1.1 M	5.3M	265	1,300	11. s	.56G	**6. s**	**.41G**
cwi_2165_8723	.22M	8.7M	4,256	20,880	20 s	.95G	**10 s**	**.71G**
vasy_6120_11031	.61M	11 M	2,505	5,358	20 s	1.4 G	20 s	**1.1 G**
vasy_2581_11442	2.6 M	11 M	704,737	3,972,600	30 s	**1.4 G**	**20 s**	1.5 G
vasy_4220_13944	4.2 M	14 M	1,186,266	6,863,329	40 s	2.1 G	**20 s**	**1.4 G**
vasy_4338_15666	4.3 M	16 M	704,737	3,972,600	50 s	1.8 G	**30 s**	**1.8 G**
cwi_2416_17605	2.2 M	16 M	730	2,899	**9. s**	1.6 G	20 s	**1.1 G**
vasy_11026_24660	11 M	25 M	775,618	2,454,834	80 s	2.9 G	**60 s**	**2.6 G**
vasy_12323_27667	12 M	28 M	876,944	2,780,022	90 s	3.3 G	**70 s**	**3.0 G**
vasy_8082_42933	8.1 M	42 M	290	680	90 s	4.5 G	**50 s**	**3.3 G**
cwi_7838_59101	7.8 M	58 M	62,031	470,230	200 s	5.9 G	**120 s**	**4.2 G**
1394-fin-vvlarge	38 M	85 M	607,942	1,590,210	300 s	10 G	**200 s**	**8.1 G**
cwi_33949_165318	34 M	165 M	12,463	71,466	500 s	1.8 G	500 s	**1.3 G**

7 Benchmarks

In Table 1 we show some benchmarks illustrating the performance of the new algorithm. The benchmarks are mainly taken from the VLTS benchmark set[1]. We show the number of states and transitions after removal of τ-cycles under the header "After SCC". In the next columns the number of states after branching bisimulation reduction are given. Subsequently, the time and memory required to apply branching bisimulation algorithms are given where the time is taken to calculate the branching bisimulation equivalence classes starting with the τ-loop reduced transition systems. We compare the fastest known branching bisimulation algorithm headed "JGKW" with the algorithm presented in this paper. The row starting with 1394-fin-vvlarge reports on reducing the state space obtained from the mCRL2 model of the 1394 firewire protocol with 20 data elements as described in [3].

The results are obtained on a computer with an Intel Xeon Gold 6136 CPU at 3.00GHz processors and plenty of memory. Every benchmark was run 30 times. We removed the five fastest and the five slowest times and report the average of the remaining 20. Times are rounded to significant digits; trailing zeroes not followed by a decimal point are insignificant. The algorithm in this paper has almost always a better time and memory performance than JGKW. We also implemented optimisations, one of which performs initialisation without BLC sets. Never being detrimental, this further reduces the running time up to a factor 2. But as they are not in line with the presented algorithm we decided not to report them.

In both compared implementations states and label indices are stored as 64-bit numbers. We can substantially cut down on memory requirements by compressing these encodings, as is for instance done in [9], where incidentally an algorithm for strong bisimulation is compared to an algorithm for branching bisimulation applied to calculate a strong bisimulation reduction. As such comparisons are not very informative, we took care that the implementations of the two algorithms use the same implementation style. Both compared algorithms, along

[1] `http://cadp.inria.fr/resources/vlts`.

with a number of others, can be tried in the freely available mCRL2 toolset [6] available at `https://www.mcrl2.org` and are part of the summer 2025 release. This toolset is open-source, which means that that the complex implementation code for these algorithms can be extracted and used elsewhere if so desired.

References

1 Rocco De Nicola and Frits Vaandrager. Action versus state based logics for transition systems. In I. Guessarian, editor, *Semantics of systems of concurrent processes: LITP spring school on theoretical computer science*, volume 469 of *LNCS*, pages 407–419. Springer, Berlin, 1990. doi:10.1007/3-540-53479-2_17.

2 Jean-Claude Fernandez. An implementation for an efficient algorithm for bisimulation equivalence. *Science of Computer Programming*, 13(2–3):219–236, 1990. doi:10.1016/0167-6423(90)90071-K.

3 Hubert Garavel and Bas Luttik. Four formal models of IEEE 1394 link layer. In Frédéric Lang and Matthias Volk, editors, *Proceedings Sixth Workshop on Models for Formal Analysis of Real System [MARS]*, volume 399 of *EPTCS*, pages 21–100, 2024. doi:10.4204/EPTCS.399.5.

4 Rob J. van Glabbeek and W. Peter Weijland. Branching time and abstraction in bisimulation semantics. *J. ACM*, 43(3):555–600, 1996. doi:10.1145/233551.233556.

5 Jan Friso Groote, David N. Jansen, Jeroen J.A. Keiren, and Anton J. Wijs. An $O(m \log n)$ algorithm for computing stuttering equivalence and branching bisimulation. *ACM Trans. Comput. Logic*, 18(2):Article 13, 2017. doi:10.1145/3060140.

6 Jan Friso Groote and Mohammad Reza Mousavi. *Modeling and Analysis of Communicating Systems*. MIT Press, 2014. URL: `https://mitpress.mit.edu/9780262547871/`.

7 Jan Friso Groote and Frits Vaandrager. An efficient algorithm for branching bisimulation and stuttering equivalence. In M. S. Paterson, editor, *Automata, languages and programming [ICALP]*, volume 443 of *LNCS*, pages 626–638. Springer, Berlin, 1990. doi:10.1007/BFb0032063.

8 Jan Friso Groote and Anton Wijs. An $O(m \log n)$ algorithm for stuttering equivalence and branching bisimulation. In Marsha Chechik and Jean-François Raskin, editors, *Tools and algorithms for the construction and analysis of systems: TACAS*, volume 9636 of *LNCS*, pages 607–624. Springer, Berlin, 2016. doi:10.1007/978-3-662-49674-9_40.

9 Jules Jacobs and Thorsten Wißmann. Fast coalgebraic bisimilarity minimization. *Proc. ACM Program. Lang.*, 7(POPL):1514–1541, 2023. doi:10.1145/3571245.

10 David N. Jansen, Jan Friso Groote, Jeroen J.A. Keiren, and Anton Wijs. An $O(m \log n)$ algorithm for branching bisimilarity on labelled transition systems. In Armin Biere and David Parker, editors, *Tools and Algorithms for the Construction and Analysis of Systems: TACAS, Part II*, volume 12079 of *LNCS*, pages 3–20. Springer, Cham, 2020. doi:10.1007/978-3-030-45237-7_1.

11 Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *LNCS*. Springer, Berlin, 1980. doi:10.1007/3-540-10235-3.

12 Robert Paige and Robert E. Tarjan. Three partition refinement algorithms. *SIAM J. Comput.*, 16(6):973–989, 1987. doi:10.1137/0216062.

13 Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972. doi:10.1137/0201010.

14 Antti Valmari. Bisimilarity minimization in $O(m \log n)$ time. In Giuliana Franceschinis and Karsten Wolf, editors, *Applications and Theory of Petri Nets: PETRI NETS*, volume 5606 of *LNCS*, pages 123–142. Springer, Berlin, 2009. doi:10.1007/978-3-642-02424-5_9.

Time for Timed Monitorability

Thomas M. Grosen
Aalborg University, Aalborg, Denmark

Sean Kauffman
Queen's University, Kingston, Canada

Kim G. Larsen
Aalborg University, Aalborg, Denmark

Martin Zimmermann
Aalborg University, Aalborg, Denmark

Abstract

Monitoring is an important part of the verification toolbox, in particular in situations where exhaustive verification using, e.g., model-checking is infeasible. The goal of online monitoring is to determine the satisfaction or violation of a specification during runtime, i.e., based on finite execution prefixes. However, not every specification is amenable to monitoring, e.g., properties for which no finite execution can witness satisfaction or violation. Monitorability is the question of whether a given specification is amenable to monitoring, and has been extensively studied in discrete time.

Here, we study the monitorability problem for real-time properties expressed as Timed Automata. For specifications given by deterministic Timed Muller Automata, we prove decidability while we show that the problem is undecidable for specifications given by nondeterministic Timed Büchi automata.

Furthermore, we refine monitorability to also determine bounds on the number of events as well as the time that must pass before monitoring the property may yield an informative verdict. We prove that for deterministic Timed Muller automata, such bounds can be effectively computed. In contrast we show that for nondeterministic Timed Büchi automata such bounds are not computable.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Modal and temporal logics

Keywords and phrases Monitorability, Monitoring, Timed Automata, MITL

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2025.19

Related Version *arXiv Version*: `https://arxiv.org/abs/2504.10008` [22]

Funding T. M. Grosen and K. G. Larsen have been funded by the Villum Investigator Grant S4OS. T. M. Grosen, K. G. Larsen, and M. Zimmermann have been supported by DIREC – Digital Research Centre Denmark. T. M. Grosen has been supported by Mitacs. S. Kauffman has been supported by the Natural Sciences and Engineering Research Council of Canada.

Acknowledgements We would like to thank Corto Mascle, who brought Rampersad et al.'s work on the complexity of suffix-universality [30] to our attention, which inspired our undecidability proof for weak monitorability.

1 Introduction

A fundamental challenge in Runtime Verification (RV) is that many properties provide no utility when monitored in an online setting. Thus, it is of utmost importance to identify those properties that do provide useful information.

Behaviors of long-running systems are typically specified as languages of infinite words, but online monitors only observe finite (prefixes of) system executions. Thus, a monitor has to determine whether such a finite prefix already implies satisfaction or violation of the

© Thomas M. Grosen, Sean Kauffman, Kim G. Larsen, and Martin Zimmermann;
licensed under Creative Commons License CC-BY 4.0

property. While many different types of monitors have been proposed, most online monitors return information in the form of *verdicts* about the finite prefix. Inherently to the problem, there are at least three verdicts [10]: $\{\top, \bot, ?\}$, where $\top$ and $\bot$ are *conclusive* verdicts that mean that the finite prefix guarantees that *every* possible infinite extension satisfies, respectively violates, the property, and the *inconclusive* verdict $?$ signifying that neither is the case.

For example, consider an arbiter granting access to a shared resource. The property φ_1 expressing "there is no request during the first second" is satisfied if the first request arrives after two seconds, no matter how the execution continues. Hence, the verdict for such a finite execution is $\top$. Similarly, the property φ_2 expressing "after any request, there are no requests for at least one second" is violated as soon as two consecutive requests are observed within one second, no matter how the execution continues. Hence, the verdict for such a finite execution is $\bot$. On the other hand, if requests arrive with a gap of two seconds between them, then the verdict for such a prefix (w.r.t. φ_2) is $?$, since there are infinite extensions satisfying the property and infinite extensions violating it.

As seen above, there are prefixes for which we get a conclusive verdict w.r.t. φ_2. This distinguishes it from properties like φ_3 expressing "every request is eventually granted", for which every finite prefix can be extended to satisfy the property and can be extended to violate it. Hence, every finite prefix yields the verdict $?$. Phrased concisely: monitoring the property φ_3 is futile.

However, for the user it is not transparent, while receiving the verdict $?$, whether in the future a conclusive verdict may be given, or whether every possible extension yields the verdict $?$. The concept of *monitorability* has been introduced to capture those properties that are amenable to monitoring. It comes in two variants, strong monitorability (every prefix can be extended to one that yields a conclusive verdict) and weak monitorability (some prefix yields a conclusive verdict). In this language, φ_3 is not weakly monitorable while φ_1 and φ_2 are (even strongly) monitorable. Thus, before constructing and deploying a monitor for a property, it is prudent to first check whether the property is monitorable.

The problem of deciding if a property is monitorable has been studied extensively over the last 20 years (we discuss related work in Section 7), but only in the setting of discrete time until very recently. However, many properties require real-time constraints to express, e.g., deadlines like "every request is answered within 745 ms". In particular, safety-critical systems are nearly always real-time systems with physics-based deadlines, and these systems tend to benefit the most from formal verification methods like RV. Had the property "the Therac 25 control program must wait eight seconds before switching between X-ray and electron modes" been monitored, it might have saved lives [26]. While monitoring algorithms exist for real-time properties [34, 9, 8, 23], the problem of real-time monitorability has gone largely unexamined.

Our Contribution. This work makes monitoring of real-time systems more useful by examining the monitorability problem for real-time properties and by introducing qualitative refinements of the verdict $?$. We consider real-time properties expressed by Timed Automata (TA) over infinite words [3]. Nondeterministic Timed Büchi Automata (TBA) are used for model checking tools like Uppaal [25] and for monitoring temporal logics [23, 20, 14] and are strictly more expressive than deterministic Timed Muller Automata (DTMA). We prove that strong and weak monitorability are undecidable for nondeterministic TBA, but decidable for DTMA. Thus, one can algorithmically determine that it is futile to monitor properties like φ_3, thereby increasing the applicability of monitoring of real-time systems.

Furthermore, we introduce monitorability with step-bounded horizons that strengthens monitorability by limiting the number of events in a timed-word before a conclusive verdict must be reached. A step-bounded horizon allows one to determine for a given property and $n \in \mathbb{N}$, if a conclusive verdict is possible within n steps, enabling corrective actions earlier. Again, we show that monitorability with step-bounded horizons is undecidable for nondeterministic TBA, but decidable for DTMA.

Finally, we refine monitoring of real-time properties with time-horizon verdicts. Here, the verdict *?* is enhanced with information about the minimum time until a conclusive verdict may be reached. Intuitively, before this time is reached, the monitor will only yield the inconclusive verdict *?*, i.e., no information can be gained from querying the monitor before. This notion was introduced by Grosen et al. [23] as "time-predictive" monitoring. Here, we formally prove that time-horizon queries can be computed effectively for DTMA.

Thus, our results highlight the importance of properties being given by deterministic timed automata when checking monitorability. This is in contrast to monitoring itself, where it suffices to have nondeterministic automata for the property *and* its negation [23], which is, e.g., the case when specifying properties in Metric Interval Temporal Logic [4]. Finally, we show that it is necessary to have automata for the property and the complement, as the monitoring function is otherwise not effectively computable.

2 Preliminaries

The nonnegative integers are denoted by $\mathbb{N}$ and the nonnegative reals by $\mathbb{R}_{\geq 0}$. An alphabet is a finite nonempty set of letters.

A timed word is a pair $\rho = (\sigma, \tau)$, where σ is a (finite or infinite) word over an alphabet Σ and τ is a sequence of non-decreasing, non-negative real numbers of the same length as σ. For convenience, we often write $(\sigma_1, \tau_1)(\sigma_2, \tau_2)\cdots$ for a timed word (σ, τ). $T\Sigma^*$ and $T\Sigma^\omega$ denote the sets of finite and infinite timed words over Σ. For $n \in \mathbb{N} \cup \{\infty\}$ we denote by $T\Sigma^{\leq n}$ the set of timed words over Σ of length at most n. Given a finite word $\rho = (\sigma_1, \tau_1)(\sigma_2, \tau_2)\cdots(\sigma_n, \tau_n)$, we denote its duration as $\tau(\rho) = \tau_n$. Slightly abusively, we write ε for the empty timed word $(\varepsilon, \varepsilon)$ and define $\tau(\varepsilon) = 0$. For a finite timed word $\rho = (\sigma_1, \tau_1)(\sigma_2, \tau_2)\cdots(\sigma_n, \tau_n)$, a finite or infinite word $\rho' = (\sigma'_1, \tau'_1)(\sigma'_2, \tau'_2)\cdots$ and a timepoint $t \geq \tau(\rho)$, we define the concatenation of ρ and ρ' at t as

$$\rho \cdot_t \rho' = (\sigma_1, \tau_1)(\sigma_2, \tau_2)\cdots(\sigma_n, \tau_n)(\sigma'_1, \tau'_1 + t)(\sigma'_2, \tau'_2 + t)\cdots$$

which is a timed word. As a shorthand, we write $\rho \cdot \rho'$ for $\rho \cdot_{\tau(\rho)} \rho'$. Given two finite words ρ, ρ' and a timepoint $t \geq \tau(\rho)$, we write $\rho \sqsubseteq_t \rho'$ if there exists a ρ'' such that $\rho \cdot_t \rho'' = \rho'$.

▶ **Example 1.** Consider the words $\rho = (a, 0)(b, 10)$ and $\rho' = (c, 5)(d, 15)$. We have
- $\rho \cdot \rho' = \rho \cdot_{10} \rho' = (a, 0)(b, 10)(c, 15)(d, 25)$,
- $\rho \cdot_{15} \rho' = (a, 0)(b, 10)(c, 20)(d, 30)$, and
- $\rho' \cdot \rho = \rho' \cdot_{15} \rho = (c, 5)(d, 15)(a, 15)(b, 25)$.

Timed Automata. A timed Büchi automaton (TBA) is a tuple $\mathcal{A} = (Q, Q_0, \Sigma, \mathcal{X}, \Delta, F)$ where Q is a finite set of locations, $Q_0 \subseteq Q$ is the set of initial locations, Σ is an alphabet, $\mathcal{X}$ is a finite set of clocks, $F \subseteq Q$ is a set of accepting locations, and $\Delta \subseteq Q \times Q \times \Sigma \times 2^{\mathcal{X}} \times G(\mathcal{X})$ is a set of transitions, where $G(\mathcal{X})$ is the set of clock constraints over $\mathcal{X}$. A transition $(q, q', \alpha, \lambda, g) \in \Delta$ is an edge from q to q' with label $\alpha \in \Sigma$, where $\lambda \in 2^{\mathcal{X}}$ is a set of clocks to be reset and $g \in G(\mathcal{X})$ is a clock constraint. A clock constraint is a finite conjunction of atomic constraints of the form $x \sim n$ where $x \in \mathcal{X}$, $n \in \mathbb{N}$, and $\sim \in \{<, \leq, =, \geq, >\}$. A state of $\mathcal{A}$ is a pair (q, v) where $q \in Q$ and $v \colon \mathcal{X} \to \mathbb{R}_{\geq 0}$ is a clock valuation.

A run of $\mathcal{A}$ from a state (q_0, v_0) over an infinite word $(\sigma, \tau) \in T\Sigma^\omega$ is a sequence of steps of the form

$$(q_0, v_0) \xrightarrow{(\sigma_1,\tau_1)} (q_1, v_1) \xrightarrow{(\sigma_2,\tau_2)} (q_2, v_2) \xrightarrow{(\sigma_3,\tau_3)} \cdots$$

where for every $i \geq 1$, there is a transition $(q_{i-1}, q_i, \sigma_i, \lambda_i, g_i) \in \Delta$ such that $v_i(x) = 0$ for all $x \in \lambda_i$ and $v_i(x) = v_{i-1}(x) + \tau_i - \tau_{i-1}$ otherwise, and g is satisfied by $v_{i-1} + \tau_i - \tau_{i-1}$ (where we use $\tau_0 = 0$). For a run r, $\mathit{inf}(r) \subseteq Q$ denotes the set of locations visited infinitely often in r. A run r is (Büchi) accepting if $\mathit{inf}(r) \cap F \neq \emptyset$.

A timed Muller automaton (TMA) $\mathcal{A} = (Q, Q_0, \Sigma, \mathcal{X}, \Delta, \mathcal{F})$ is like a TBA, but the set F of accepting locations is replaced by a set $\mathcal{F} \subseteq 2^Q$ of sets of locations. A run r of $\mathcal{A}$ is (Muller) accepting if $\mathit{inf}(r) \in \mathcal{F}$.

The language $L(\mathcal{A})$ of a timed (Büchi or Muller) automaton is the set of words $\rho \in T\Sigma^\omega$ such that $\mathcal{A}$ has an accepting run over ρ.

An automaton is deterministic if the set of initial locations is a singleton and if all edges from the same location and with the same label must have disjoint clock constraints. We use the abbreviations DTBA and DTMA to refer to the two deterministic automaton models.

▶ **Proposition 2.** *The following results on TBA and TMA are due to Alur and Dill [3].*

1. *Let $\mathcal{A} = (Q, Q_0, \Sigma, \mathcal{X}, \Delta, F)$ be a TBA and let $\mathcal{A}' = (Q, Q_0, \Sigma, \mathcal{X}, \Delta, \{F' \mid F \cap F' \neq \emptyset\})$, which is a TMA. Then, $L(\mathcal{A}) = L(\mathcal{A}')$. Furthermore, if $\mathcal{A}$ is deterministic, then so is $\mathcal{A}'$.*
2. *For every TMA $\mathcal{A} = (Q, Q_0, \Sigma, \mathcal{X}, \Delta, \mathcal{F})$, there is a TBA $\mathcal{A}'$ with $L(\mathcal{A}) = L(\mathcal{A}')$. The set of locations of $\mathcal{A}'$ has the form $Q \times \mathcal{F} \times \{0, 1, \ldots, |Q|\}$.*
3. *The class of languages accepted by DTMA is a strict subset of the class of languages accepted by TBA.*

Logic. We use Metric Interval Temporal Logic (MITL) to formally express properties to be monitored. The syntax of MITL formulas over a finite alphabet Σ is defined as

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid X_I\varphi \mid \varphi\ U_I\varphi$$

where $p \in \Sigma$ and I ranges over non-singular intervals over $\mathbb{R}_{\geq 0}$ with endpoints in $\mathbb{N} \cup \{\infty\}$. Note that we often write $\sim\ n$ for $I = \{d \in \mathbb{R}_{\geq 0} \mid d \sim n\}$ where $\sim \in \{<, \leq, \geq, >\}$, and $n \in \mathbb{N}$. We also define the standard syntactic sugar $\texttt{true} = p \vee \neg p$, $\texttt{false} = \neg\texttt{true}$, $\varphi \wedge \psi = \neg(\neg\varphi \vee \neg\psi)$, $\varphi \rightarrow \psi = \neg\varphi \vee \psi$, $F_I\varphi = \texttt{true}\ U_I\varphi$, and $G_I\varphi = \neg F_I \neg\varphi$.

The semantics of MITL is defined over infinite timed words. Given such a timed word $\rho = (\sigma_1, \tau_1)(\sigma_2, \tau_2) \cdots \in T\Sigma^\omega$, a position $i \geq 1$, and an MITL formula φ, we inductively define the satisfaction relation $\rho, i \models \varphi$ as follows:

- $\rho, i \models p$ if and only if $p = \sigma_i$.
- $\rho, i \models \neg\varphi$ if and only if $\rho, i \not\models \varphi$.
- $\rho, i \models \varphi \vee \psi$ if $\rho, i \models \varphi$ or $\rho, i \models \psi$.
- $\rho, i \models X_I\varphi$ if and only if $\rho, (i+1) \models \varphi$ and $\tau_{i+1} - \tau_i \in I$.
- $\rho, i \models \varphi\ U_I\psi$ if and only if there exists $k \geq i$ s.t. $\rho, k \models \psi$, $\tau_k - \tau_i \in I$, and $\rho, j \models \varphi$ for all $i \leq j < k$.

We write $\rho \models \varphi$ whenever $\rho, 1 \models \varphi$, and say that ρ satisfies φ. The language $L(\varphi)$ of an MITL formula φ is the set of all infinite timed words that satisfy φ.

▶ **Proposition 3** ([4, 12])**.** *For each MITL formula φ there is a TBA $\mathcal{A}$ such that $L(\varphi) = L(\mathcal{A})$.*

▶ **Example 4.** Figure 1 illustrates the above proposition by providing a DTBA over the alphabet $\Sigma = \{a, b, c\}$ for the formula $F_{[0,10]}a \wedge G_{[0,20]}\neg b$ and its negation.

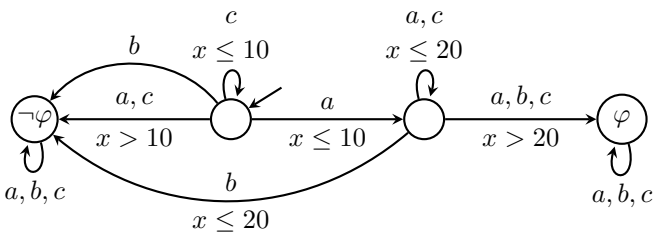


Figure 1 DTBA for the language of the MITL formula $\varphi = F_{[0,10]}a \wedge G_{[0,20]}\neg b$ and its negation: If location φ ($\neg\varphi$) is accepting then it accepts $L(\varphi)$ ($L(\neg\varphi)$, respectively).

Monitoring. The monitoring problem asks to make verdicts about the satisfaction or violation of properties (over infinite timed words) after having observed only a finite prefix. Here, we follow the classical approach of considering the possible extensions of a finite observations.

▶ **Definition 5** (Observation). *An observation is a pair (ρ, t) containing a finite timed word ρ and a timepoint $t \geq \tau(\rho)$, representing the current timepoint (which might be later than the last observed event in ρ). As we use observations as inputs for algorithms, we require that all timepoints in ρ and t are rational.*

We continue with giving some intuition for the three-valued monitoring approach. Here, one is given an observation and aims to determine whether a property φ (of infinite timed words) is already satisfied, already violated, or neither.

- If all possible extensions of the observation satisfy φ, then we give the corresponding verdict $\top$ signifying that the observation conclusively witnesses satisfaction of φ.
- If all possible extensions of the observation violate φ, then we give the corresponding verdict $\bot$ signifying that the observation conclusively witnesses violation of φ.
- Otherwise, i.e., if there is an extension of the observation that satisfies φ and there is a extension of the observation that violates φ, then we give the inconclusive verdict $?$.

Let us formalize this intuition.

▶ **Definition 6** (Timed Monitoring Function). *Given a property $\varphi \subseteq T\Sigma^\omega$ and an observation (ρ, t), the monitoring function V_φ is defined as*

$$V_\varphi(\rho, t) = \begin{cases} \top & \text{if } \rho \cdot_t \mu \in \varphi \text{ for all } \mu \in T\Sigma^\omega, \\ \bot & \text{if } \rho \cdot_t \mu \notin \varphi \text{ for all } \mu \in T\Sigma^\omega, \\ ? & \text{otherwise.} \end{cases}$$

In the following, we use $V_\varphi(\rho)$ as a shorthand for $V_\varphi(\rho, \tau(\rho))$.

▶ **Example 7.** Consider the specification $\varphi = F_{[0,10]}a \wedge G_{[0,20]}\neg b$ from Example 4. We have

- $V_\varphi((a,3),4) = ?$,
- $V_\varphi((a,11),11) = \bot$,
- $V_\varphi((a,3)(c,7),13) = ?$,
- $V_\varphi((a,3)(c,7)(c,22),22) = \top$, but
- $V_\varphi((a,3)(c,7)(b,12),12) = \bot$. Also,
- $V_\varphi((a,3)(c,7),22) = \top$ while
- $V_\varphi((a,3)(c,7)) = V_\varphi((a,3)(c,7),7) = ?$, i.e., the current timepoint t can yield conclusive verdicts when time is passing, even if no new events are observed.

▶ **Proposition 8** (Effectiveness of Timed Monitoring [23]). *V_φ is effectively computable by a zone-based online algorithm*[1] *that requires TBA for* both *φ and $T\Sigma^\omega \setminus \varphi$.*

Recall that TBA are *not* closed under complement [3], so this result is not applicable to all properties accepted by TBA. However, for the important special case of properties φ specified in MITL, V_φ is effectively computable, as MITL properties are closed under complementation (as the logic allows for negations) and MITL formulas can be translated into equivalent TBA (see Proposition 3). Similarly, when φ is given by a DTMA, then V_φ is effectively computable, as DTMA are closed under complementation and can be turned into equivalent TBA [3].

However, it was previously open whether V_φ was computable if φ was given by a non-deterministic automaton, but we did not have access to an automaton for the complement $T\Sigma^\omega \setminus \varphi$. In Section 3, we answer the question negatively.

Monitorability. Not every property is amenable to monitoring, e.g., for $\varphi = G_{\geq 0}F_{\geq 0}a$, we have $V_\varphi(\rho, t) = ?$ for every observation (ρ, t). The reason is that every ρ can be extended to satisfy φ and can be extended to violate φ. In the untimed setting, much effort has been put into characterizing the monitorable properties, i.e., those for which monitoring can generate some information. Here, we consider monitorability in the timed setting.

▶ **Definition 9** (Timed Monitorability). *Fix an observation (ρ, t) and a property $\varphi \subseteq T\Sigma^\omega$.*

- *φ is strongly (ρ, t)-monitorable if and only if*

  for all $\rho' \in T\Sigma^*$ there exists $\rho'' \in T\Sigma^*$ such that $V_\varphi(\rho \cdot_t \rho' \cdot \rho'') \in \{\top, \bot\}$.

- *φ is weakly (ρ, t)-monitorable if and only if*

  there exists $\rho'' \in T\Sigma^*$ such that $V_\varphi(\rho \cdot_t \rho'') \in \{\top, \bot\}$.

- *φ is strongly monitorable if it is strongly $(\varepsilon, 0)$-monitorable.*
- *φ is weakly monitorable if it is weakly $(\varepsilon, 0)$-monitorable.*

▶ **Example 10.**

1. Consider the property $\varphi_1 = F_{\geq 0}a$. For every observation (ρ, t), we have $V_{\varphi_1}(\rho \cdot_t (a, 0), t) = \top$. Hence, φ_1 is strongly monitorable.
2. Now, consider $\varphi_2 = a \to G_{\geq 0}F_{\geq 0}a$. Then, we have $V_{\varphi_2}((b, 0), 0) = \top$, as every extension of $((b, 0), 0)$ satisfies φ_2 (as the premise is violated). Hence, φ_2 is weakly monitorable. However, it is not strongly monitorable: Consider the observation $((a, 0), 0)$, for which every extension satisfies the premise. We have $(a, 0) \cdot_0 \rho'' \cdot (a, 0)(a, 1)(a, 2) \cdots \models \varphi_2$ and $(a, 0) \cdot_0 \rho'' \cdot (b, 0)(b, 1)(b, 2) \cdots \not\models \varphi_2$. Hence, $V_{\varphi_2}((a, 0) \cdot_0 \rho'') = ?$ for all ρ''.
3. Now, consider $\varphi_3 = G_{\geq 0}F_{\geq 0}a$. Arguments as for φ_2 show that it is neither strongly nor weakly monitorable, as every finite word can be extended by $(a, 0)(a, 1)(a, 2) \cdots$ to satisfy φ_3 and can be extended by $(b, 0)(b, 1)(b, 2) \cdots$ to violate φ_3.

▶ Remark 11. The astute reader might wonder why we quantify only over words ρ' (and ρ'') in Definition 9 and not over words and timepoints to concatenate at. The reason is that both definitions are equivalent, as $\rho_1 \cdot_t \rho_2$, for finite words ρ_1 and ρ_2 and a timepoint $t \geq \tau(\rho)$, is equal to $\rho_1 \cdot \rho_2'$, where ρ_2' is obtained from ρ_2 by incrementing all its timepoints by $t - \tau(\rho_1)$.

Let us continue by collecting some simple consequences of Definition 9.

[1] See Page 11 for a formal definition of zones.

▶ Remark 12. Let (ρ, t) and (ρ', t') be two observations with $\rho \sqsubseteq_t \rho'$, and let $\varphi \subseteq T\Sigma^\omega$.

1. If φ is strongly (ρ, t)-monitorable, then it is also weakly (ρ, t)-monitorable. Thus, if φ is strongly monitorable, then it is also weakly monitorable.
2. If φ is strongly (ρ, t)-monitorable, then it is also strongly (ρ', t')-monitorable.
3. If φ is not weakly (ρ, t)-monitorable, then it is also not weakly (ρ', t')-monitorable.

In the following we study the decidability of monitorability, i.e., we consider the following decision problems where properties are given by timed automata:

1. Given a property φ, is φ strongly monitorable?
2. Given a property φ, is φ weakly monitorable?
3. Given a property φ and an observation (ρ, t), is φ strongly (ρ, t)-monitorable?
4. Given a property φ and an observation (ρ, t), is φ weakly (ρ, t)-monitorable?

By definition, if Problem 3 is decidable for a class of properties, then Problem 1 is also decidable for the same class of properties. Hence, if Problem 1 is undecidable for a class of properties, then Problem 3 is also undecidable for the same class of properties. A similar relation holds between Problem 2 and Problem 4.

3 Monitoring and Monitorability for TBA

In this section, we prove that the monitoring function is not computable and that the strong types of monitorability are undecidable when the property is given by a TBA. Note that this shows that the positive results for monitoring in the literature [23, 20, 14], which require automata both for the property and its complement, are tight in that sense: Only giving an automaton for the property, but not for its complement, is not sufficient to compute the monitoring function. We start by investigating the computability of the monitoring function.

▶ **Theorem 13** (Ineffectiveness of Timed Monitoring). *The function "Given a TBA $\mathcal{A}$ and an observation (ρ, t), return $V_{L(\mathcal{A})}(\rho, t)$" is not computable.*

Proof. For every property $\varphi \subseteq T\Sigma^\omega$, we have $V_\varphi(\varepsilon, 0) = \top$ if and only if $\varphi = T\Sigma^\omega$. Hence, universality of a TBA $\mathcal{A}$ reduces to checking whether $V_{L(\mathcal{A})}(\varepsilon, 0) = \top$. As universality for timed automata is undecidable [5], the monitoring function cannot be computable. ◀

Note that the specification automaton $\mathcal{A}$ is part of the input in the problem considered in Theorem 13. We leave it open whether $V_{L(\mathcal{A})}$ is computable for every fixed $\mathcal{A}$, i.e., in the setting where only the observation is the input.

Next, we turn our attention to deciding monitorability for TBA.

▶ **Theorem 14.** *Strong and weak monitorability are undecidable for properties given by TBA.*

Proof. Strong monitorability for untimed non-deterministic Büchi Automata is PSPACE-complete [17], which can be shown by a reduction from the universality problem [18]. Analogously, we reduce the (undecidable [3]) universality problem for non-deterministic timed automata (over finite words) to the problem of strong monitorability, following Diekert, Muscholl, and Walukiewicz [18].

Let $\mathcal{A} = (Q, Q_0, \Sigma, \mathcal{X}, \Delta, F)$ be such a timed automaton, i.e., a finite run is accepting if it ends in a location in F. We assume w.l.o.g. that $\mathcal{A}$ is complete in the sense that every word has a run. This can always be achieved by adding a fresh sink location and by rerouting all missing transitions to it.

We add a new letter $\# \notin \Sigma$ to obtain $\Sigma_\# = \Sigma \cup \{\#\}$ and construct a TBA $\mathcal{A}' = (Q', Q_0, \Sigma_\#, \mathcal{X}, \Delta', F')$ from $\mathcal{A}$ such that $\mathcal{A}'$ is strongly monitorable if and only if $L(\mathcal{A}) = T\Sigma^*$. To this end, we introduce three new locations r, s, t, i.e., $Q' = Q \cup \{r, s, t\}$. Next, we define Δ' (see Figure 2) by copying the transitions from Δ and by adding the following transitions, where we write $q \xrightarrow{a} q'$ to denote $(q, q', a, \emptyset, \texttt{true})$):

- $\{q \xrightarrow{\#} r \mid q \in Q \setminus F\}$: from every non-accepting location of $\mathcal{A}$, there is a #-transition to r.
- $\{r \xrightarrow{a} s, s \xrightarrow{a} s \mid a \in \Sigma\}$: for every $a \neq \#$ there is an a-transition from r to s and an a-labeled self-loop on s.
- $\{s \xrightarrow{\#} r, r \xrightarrow{\#} r\}$: there is a #-transition from s to r and a #-labeled self-loop on r.
- $\{q \xrightarrow{\#} t \mid q \in F\}$: from every accepting location there is a #-transition to t.
- $\{t \xrightarrow{a} t \mid a \in \Sigma_\#\}$: for every letter in $\Sigma_\#$, there is a self-loop on t labeled with that letter.

We define the accepting locations of $\mathcal{A}'$ as $F' = Q \cup \{r, t\}$.

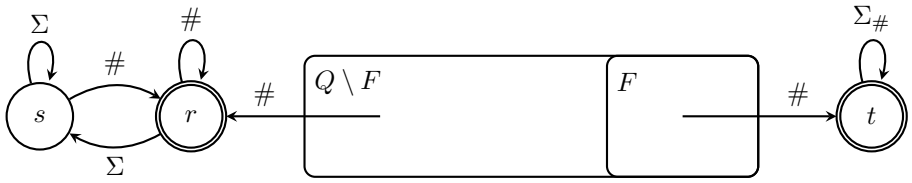


Figure 2 The TBA $\mathcal{A}'$ constructed for the proof of Theorem 14.

Figure 2 shows the TBA $\mathcal{A}'$ with the locations of the (finite-word) timed automaton $\mathcal{A}$ partitioned into F and $Q \setminus F$. In the figure, the new locations r, s, and t and accompanying transitions are shown separately from the original automaton. Double circles denote accepting locations. Intuitively, processing a # from an accepting location of $\mathcal{A}$ takes the run to the accepting sink t. On the other hand, processing a # from a non-accepting location of $\mathcal{A}$ takes the run to a component where the run continuation is accepting if and only if it processes infinitely many #'s.

It remains to show that the language $L(\mathcal{A}')$ is strongly monitorable if and only if $L(\mathcal{A}) = T\Sigma^*$. One direction is trivial: If $L(\mathcal{A}) = T\Sigma^*$, then $L(\mathcal{A}') = T\Sigma^\omega_\#$, as all words without a # can be processed using the states of the complete automaton $\mathcal{A}$ (which are all accepting in $\mathcal{A}'$) and all words with a # can be accepted by moving to state t (which is an accepting sink) when processing the first #. Hence, $L(\mathcal{A}')$ is strongly monitorable.

On the other hand, if $L(\mathcal{A}) \neq T\Sigma^*$, then there exists a finite word $\rho \notin L(\mathcal{A})$. Hence, every run prefix of $\mathcal{A}'$ processing $\rho \cdot_{\tau(\rho)} (\#, \tau(\rho))$ must be in location r. Now, chose some $a \in \Sigma$. Then, for all $\rho'' \in T\Sigma^*_\#$, we have:

- $\rho \cdot_{\tau(\rho)} (\#, \tau(\rho)) \cdot \rho'' \cdot (a,0)(a,1)(a,2)\cdots \notin L(\mathcal{A}')$ (as it contains only finitely many #).
- $\rho \cdot_{\tau(\rho)} (\#, \tau(\rho)) \cdot \rho'' \cdot (\#,0)(\#,1)(\#,2)\cdots \in L(\mathcal{A}')$ (as it contains infinitely many #).

Hence, $V_{L(\mathcal{A}')}(\rho \cdot_{\tau(\rho)} (\#, \tau(\rho)) \cdot \rho'') = ?$ for all ρ''. Thus, $L(\mathcal{A}')$ is not strongly monitorable.

In the case for weak monitorability we reduce the (undecidable [3]) universality problem for TBA (not TA as we did in the strong case). Additionally, as an intermediate step, we consider a problem about Brzozowski derivatives of properties of infinite timed words: Let $\rho \in T\Sigma^*$ be a finite timed word and $\varphi \subseteq T\Sigma^\omega$ be such a property. We say that ρ is a universal prefix for φ if the Brzozowski derivative

$$\{\mu \in T\Sigma^\omega \mid \rho \cdot \mu \in \varphi\}$$

of φ and ρ is equal to $T\Sigma^\omega$. Note that if φ is equal to $T\Sigma^\omega$, then every prefix is universal for φ. However, the property $(a,0) \cdot T\Sigma^\omega$ has a universal prefix (e.g., $(a,0)$), but is not universal.

Further, we say that φ is weakly $\top$-monitorable (cp. Definition 21) if and only if there is a ρ such that $V_\varphi(\rho) = \top$, i.e., in comparison to (standard) weak monitorability, we only consider the verdict $\top$. Note that φ has a universal prefix if and only if it is weakly $\top$-monitorable.

So, it remains to first reduce universality of TBA to the existence of a universal prefix for languages of TBA and then to reduce weak $\top$-monitorability to weak monitorability.

We begin with reducing TBA universality to the existence of universal prefixes. Intuitively, we add edges allowing to "restart" a run which allows every prefix to simulate the empty prefix. Then, every prefix behaves like the empty prefix, which is universal if and only if the automaton is universal.

Let $\mathcal{A} = (Q, Q_0, \Sigma, \mathcal{X}, \Delta, F)$ be a TBA. We add a new letter $\# \notin \Sigma$ to obtain $\Sigma_\# = \Sigma \cup \{\#\}$ and construct a TBA $\mathcal{A}' = (Q', Q_0, \Sigma_\#, \mathcal{X}, \Delta', F')$ from $\mathcal{A}$ such that $L(\mathcal{A}) = T\Sigma^\omega$ if and only if $\mathcal{A}'$ has a universal prefix. To this end, we introduce two new locations r and s, i.e., $Q' = Q \cup \{r, s\}$. Next, we define Δ' (see Figure 3) by copying the transitions from Δ and by adding the following transitions, where we write $q \xrightarrow{a} q'$ to denote $(q, q', a, \emptyset, \texttt{true})$:

- $\{q \xrightarrow{a} r \mid q \in Q_0, a \in \Sigma_\#\}$: from every initial location of $\mathcal{A}$, there is an a-transition to r for every $a \in \Sigma_\#$.
- $\{r \xrightarrow{a} s, s \xrightarrow{a} s \mid a \in \Sigma\}$: for every $a \neq \#$ there is an a-transition from r to s and an a-labeled self-loop on s.
- $\{s \xrightarrow{\#} r, r \xrightarrow{\#} r\}$: there is a $\#$-transition from s to r and a $\#$-labeled self-loop on r.
- $\{(q, q_0, \#, \mathcal{X}, \texttt{true}) \mid q \in Q, q \in Q_0\}$: from every location there is a $\#$-transition to each initial location, which resets all clocks.

We define the accepting locations of $\mathcal{A}'$ as $F' = Q \cup \{r\}$.

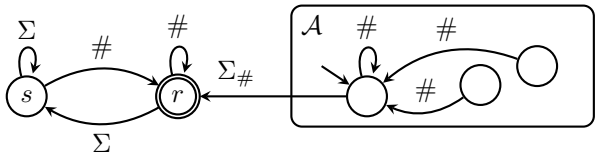


Figure 3 The TBA $\mathcal{A}'$ constructed for the proof of Theorem 14.

Let $L(\mathcal{A}) = T\Sigma^\omega$ and fix some $\mu' \in T\Sigma_\#^\omega$. If μ' contains infinitely many #'s, then it is accepted by $\mathcal{A}'$ using the new states r and s. On the other hand, if μ' contains only finitely many #'s, then it has the form

$$\mu' = \rho_0 \cdot (\#, t_0) \cdot \rho_1 \cdot (\#, t_1) \cdots (\#, t_{n-2}) \cdot \rho_{n-1} \cdot (\#, t_{n-1}) \cdot \mu''$$

for some $n \geq 0$, where the ρ_i and μ'' are #-free. As each ρ_i is a prefix of some word in $L(\mathcal{A}) = T\Sigma^\omega$ and μ'' is in $L(\mathcal{A}) = T\Sigma^\omega$, one can construct an accepting run of $\mathcal{A}'$ on μ'. Thus, $L(\mathcal{A}') = T\Sigma_\#^\omega$, i.e., ε is a universal prefix of $\mathcal{A}'$.

Now for the other direction. If a word ρ is a universal prefix of $\mathcal{A}'$, then for all $\mu \in T\Sigma^\omega$, the word $\rho \cdot (\#, 0) \cdot \mu$ is accepted by $\mathcal{A}'$. Hence, there is an accepting run of $\mathcal{A}'$ on $\rho \cdot (\#, 0) \cdot \mu$. As all #-transitions lead to an initial state of $\mathcal{A}$ and reset the clocks, and as μ does not contain any #'s, the suffix of the run on μ must also be an accepting run of $\mathcal{A}$, i.e., we have $\mu \in L(\mathcal{A})$. Thus, $L(\mathcal{A}) = T\Sigma^\omega$. This concludes the first step of our proof.

In the second and last step of our proof we reduce weak $\top$-monitorability to (standard) weak monitorability. Intuitively, we manipulate the automaton so that it can never give the verdict $\bot$ while preserving the existence of observations that yield the verdict $\top$.

Let $\mathcal{A} = (Q, Q_0, \Sigma, \mathcal{X}, \Delta, F)$ be a TBA. We add a new letter $\# \notin \Sigma$ to obtain $\Sigma_\# = \Sigma \cup \{\#\}$ and construct a TBA $\mathcal{A}' = (Q', Q_0', \Sigma_\#, \mathcal{X}, \Delta', F')$ from $\mathcal{A}$ such that $\mathcal{A}$ is weakly $\top$-monitorable if and only if $\mathcal{A}'$ is weakly monitorable.

Now Q' essentially contains two copies of Q: $Q_1 = Q \times \{1\}$ and $Q_2 = Q \times \{2\}$ as well as a new additional accepting location r. The transition relation of $\mathcal{A}'$ is defined as follows (see also Figure 4), we write $q \xrightarrow{a} q'$ to denote $(q, q', a, \emptyset, \texttt{true})$:

- $\{(q,1) \xrightarrow{\#} (q,1), (q,1) \xrightarrow{\#} r \mid q \in Q\}$: locations of Q_1 have a #-labeled self-loop and a #-transition to r.
- $\{((q,1),(q',2),a,\lambda,g) \mid (q,q',a,\lambda,g) \in \Delta\}$: locations of Q_1 have their Σ-labeled transitions redirected to Q_2.
- $\{(q,2),(q',2),a,\lambda,g) \mid (q,q',a,\lambda,g) \in \Delta\}$: locations of Q_2 have copies of the Σ-labeled transitions from $\mathcal{A}$.
- $\{(q,2) \xrightarrow{\#} (q,1) \mid q \in Q\}$: locations of Q_2 have #-labeled transitions directed back to Q_1.
- $\{r \xrightarrow{\#} r\}$: The new location r has a #-labeled self-loop.

The set Q'_0 of initial locations of $\mathcal{A}'$ is $Q_0 \times \{1\}$ and the set F' of accepting locations of $\mathcal{A}'$ is $F \times \{2\} \cup \{r\}$.

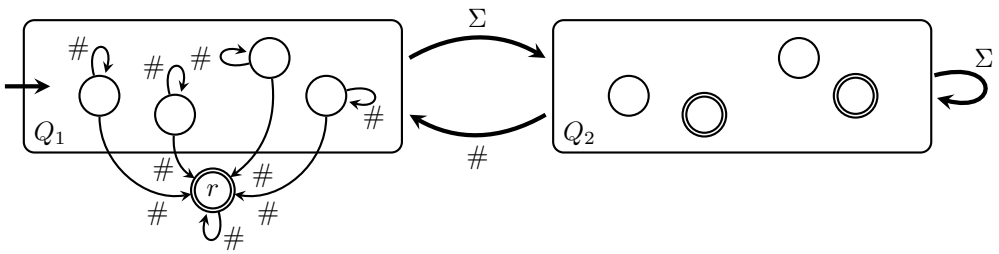


Figure 4 The TBA $\mathcal{A}'$ constructed for the proof of Theorem 14. Intuitively, $\mathcal{A}'$ has two disjoint copies of the locations of $\mathcal{A}$ and Σ-labeled transitions lead from both copies to the second copy, while #-labeled transitions from the second copy lead to the first copy. The first copy additionally has #-labeled self-loops on all locations and #-labeled transitions to r.

Now, let $\rho \in T\Sigma^*$ be a timed word witnessing weak $\top$-monitorability of $\mathcal{A}$, i.e., $\rho \cdot \mu \in L(\mathcal{A})$ for every $\mu \in T\Sigma^\omega$. We define $\rho' = \rho$. In $\mathcal{A}$, processing ρ' leads to a location in Q_2. We argue that ρ' witnesses weak monitorability of $\mathcal{A}'$. To this end, consider any $\mu' \in T\Sigma^\omega_\#$ and let μ be obtained from μ' by removing all occurrences of #. If μ is infinite, we may simply mimic the accepting run of $\mathcal{A}$ on μ from ρ in $\mathcal{A}'$. Clearly, this will also be accepting. If μ is finite, μ' must have a suffix of the form $(\#^\omega, \tau)$, where τ is a sequence of timepoints. This suffix is accepted by utilizing the # transitions to r, from where the suffix $(\#^\omega, \tau)$ can be accepted.

For the opposite direction, let $\rho' \in T\Sigma^*_\#$ be a timed word witnessing weak monitorability of $\mathcal{A}'$. That is, either $\rho' \cdot \mu' \in L(\mathcal{A}')$ for all μ' or $\rho' \cdot \mu' \notin L(\mathcal{A}')$ for all μ'. However, note that for any finite ρ', we have $\rho' \cdot (\#,0)(\#,1)(\#,2)\cdots \in L(\mathcal{A}')$, due to the #-labeled transition to r, from where $(\#,0)(\#,1)(\#,2)\cdots$ is accepted. Thus, we must be in the first case where we have $\rho' \cdot \mu' \in L(\mathcal{A}')$ for all μ'.

Now, let $\rho \in T\Sigma^*$ be obtained from ρ' by stripping all occurrences of # in ρ'. We show that ρ is a word witnessing weak $\top$-monitorability of $\mathcal{A}$. Note that any run of $\mathcal{A}'$ on ρ' reaching a location in Q_1 or Q_2 can be mimicked by a run of $\mathcal{A}$ on ρ reaching the corresponding location in Q. Let $\mu \in T\Sigma^\omega$. Then $\rho' \cdot \mu \in L(\mathcal{A}')$. Clearly, this must be due to an accepting run where a suffix of the run processing μ stays in Q_2. Hence, $\rho \cdot \mu \in L(\mathcal{A})$, as we may mimic, for the prefix ρ, the prefix processing ρ' of the accepting run of $\mathcal{A}'$ processing $\rho' \cdot \mu$, and, for the suffix μ, we simply copy the run staying in Q_2. ◀

Due to Remark 12, we also obtain the undecidability of strong and weak (ρ, t)-monitorability.

▶ **Corollary 15.** *Strong and weak (ρ, t)-monitorability is undecidable for properties given by TBA, even if (ρ, t) is fixed.*

4 Monitorability for DTMA

In this section, we show that (strong and weak) monitorability is decidable for properties given by DTMA. The key difference to TBA, for which we have shown that monitorability is undecidable, is that they are trivially closed under complement [3], which we rely on in our proof. This again demonstrates the importance of having automata for the property and its complement, as noted throughout this paper.

Let us begin by introducing some notation and definitions. Fix some TBA or TMA $\mathcal{A} = (Q, Q_0, \Sigma, \mathcal{X}, \Delta, \text{Acc})$. We write $(q_0, v_0) \xrightarrow{\rho}_{\mathcal{A}} (q_n, v_n)$ for a finite timed word $\rho = (\sigma, \tau) \in T\Sigma^*$ to denote the existence of a finite sequence

$$(q_0, v_0) \xrightarrow{(\sigma_1,\tau_1)} (q_1, v_1) \xrightarrow{(\sigma_2,\tau_2)} \cdots \xrightarrow{(\sigma_n,\tau_n)} (q_n, v_n)$$

of states, where for all $1 \leq i \leq n$ there is a transition $(q_{i-1}, q_i, \sigma_i, \lambda_i, g_i)$ such that $v_i(c) = 0$ for all c in λ_i and $v_{i-1}(c) + (\tau_i - \tau_{i-1})$ otherwise, and g is satisfied by the valuation $v_{i-1} + (\tau_i - \tau_{i-1})$, where we use $\tau_0 = 0$.

▶ **Definition 16** (Non-Empty Language States, Reach Set, *Pre**). *Let $\mathcal{A}$ be a TBA or TMA.*

- *The set of non-empty language states of $\mathcal{A}$ is*

$$S^{ne}_{\mathcal{A}} = \{s \mid s \text{ is a state of } \mathcal{A} \text{ and } L(\mathcal{A}, s) \neq \emptyset\}.$$

  Here, $L(\mathcal{A}, s)$ is the set of infinite timed words accepted by a run starting in the state s.
- *Given an observation (ρ, t), the set of states in which a run over ρ can end starting from the initial states of $\mathcal{A}$ after time t has passed is*

$$\mathcal{T}_{\mathcal{A}}(\rho, t) = \bigcup_{q_0 \in Q_0} \{(q, v + (t - \tau(\rho))) \mid (q_0, v_0) \xrightarrow{\rho}_{\mathcal{A}} (q, v)\},$$

  where v_0 is the clock valuation mapping every clock to 0. We call $\mathcal{T}_{\mathcal{A}}(\rho, t)$ the reach set of (ρ, t) in $\mathcal{A}$.
- *We define $Pre^*_{\mathcal{A}}(S)$ of a set S of states as*

$$\{(q, v) \mid (q, v) \xrightarrow{\rho}_{\mathcal{A}} (q', v') \text{ for some } (q', v' + (t - \tau(\rho))) \in S, \rho \in T\Sigma^*, \text{ and } t \geq \tau(\rho)\}.$$

A symbolic state is a pair (q, Z) of a location q and a zone Z. A zone is a finite conjunction of clock constraints of the form $x_1 \sim n$ or $x_1 - x_2 \sim n$, where x_1, x_2 are clocks, $\sim \in \{<, \leq, =, \geq, >\}$, and $n \in \mathbb{Q}$. Such a zone describes a convex set of clock valuations. Zones may be efficiently represented using so-called Difference-bounded Matrices (DBM) [11].

▶ **Proposition 17** ([23]). *There are zone-based algorithms for the following problems:*

- *Given a TBA, compute its nonempty language states.*
- *Given a TBA or TMA $\mathcal{A}$ and an observation (ρ, t), compute the reach set $\mathcal{T}_{\mathcal{A}}(\rho, t)$.*
- *Given a TBA or TMA $\mathcal{A}$ and a finite union S of symbolic states, compute $Pre^*_{\mathcal{A}}(S)$.*

We continue by showing that the non-empty language states can also be computed for TMA, relying on an analysis of Alur and Dill's translation of TMA into equivalent TBA [3]. Here, we just present the parts of their construction we need to prove our results.

▶ **Lemma 18.** *There is a zone-based algorithm that, given a TMA, computes its nonempty language states.*

Proof. Fix a TMA $\mathcal{A} = (Q, Q_0, \Sigma, \mathcal{X}, \Delta, \mathcal{F})$. The translation from TMA to TBA relies on the fact that $L(\mathcal{A}) = \bigcup_{F \in \mathcal{F}} L(\mathcal{A}_F)$, where $\mathcal{A}_F$ is the TMA $(Q, Q_0, \Sigma, \mathcal{X}, \Delta, \{F\})$. Alur and Dill translated each such $\mathcal{A}_F$ into an equivalent TBA $\mathcal{A}'_F$ with set $Q \times \{0, 1, \ldots, |F|\}$ of locations and the same clocks as $\mathcal{A}$. Then, the disjoint union of the $\mathcal{A}'_F$ for $F \in \mathcal{F}$ is a TBA that is equivalent to $\mathcal{A}$.

The TBA $\mathcal{A}'_F$ constructed by Alur and Dill satisfy the following fact: $L(\mathcal{A}'_F, ((q,0), v)) = L(\mathcal{A}_F, (q, v))$ for all locations q of $\mathcal{A}$ and all clock valuations. Thus, we have

$$S^{ne}_{\mathcal{A}} = \bigcup\nolimits_{F \in \mathcal{F}} \{(q, v) \mid ((q,0), v) \in S^{ne}_{\mathcal{A}'_F}\}.$$

Hence, we can symbolically compute the non-empty language states of $\mathcal{A}$ by symbolically computing the non-empty states of the $\mathcal{A}'_F$ and then project the states of the form $(q, 0)$ to q, but leaving the zones unchanged. ◀

Using this result, we can decide monitorability.

▶ **Theorem 19.** *Strong and weak (ρ, t)-monitorability are decidable for properties given by DTMA.*

Proof. We assume w.l.o.g. that $\mathcal{A}$ is complete in the sense that every word has a run, which then must be unique due to determinism. This can always be achieved by adding a fresh sink location and by rerouting all missing transitions to it (while preserving determinism).

Given $\mathcal{A}$, we compute the non-empty language states $S^{ne}_{\mathcal{A}}$. The complement of the non-empty language states is the empty language states, i.e., the set of states for which there is no accepting run starting there. Let us denote this set as $S^{\emptyset}_{\mathcal{A}} = \overline{S^{ne}_{\mathcal{A}}}$. Using backwards reachability, we compute all the possibly-empty language states, i.e., states from which there is at least one non-accepting run, i.e., $S^{pe}_{\mathcal{A}} = Pre^*_{\mathcal{A}}(S^{\emptyset}_{\mathcal{A}})$. Note that $S^{\emptyset}_{\mathcal{A}} \subseteq S^{pe}_{\mathcal{A}}$. If a state s is not in $S^{pe}_{\mathcal{A}}$, then all states reachable from s have at least one accepting run.

The sets $S^{ne}_{\mathcal{A}}$, $S^{\emptyset}_{\mathcal{A}}$ and $S^{pe}_{\mathcal{A}}$ are illustrated in Fig. 5.

Figure 5 Representation of all states of an automaton $\mathcal{A}$. On the left, only the empty languages states $S^{\emptyset}_{\mathcal{A}}$ and non-empty language states $S^{ne}_{\mathcal{A}}$ are shown. On the right, the possibly empty language states $S^{pe}_{\mathcal{A}}$ are added in gray. The arrows are examples of possible transitions and the dashed arrows are examples of impossible transitions. A state in $S^{\emptyset}_{\mathcal{A}}$ has no accepting run, thus cannot reach $S^{ne}_{\mathcal{A}}$. A state in $S^{ne}_{\mathcal{A}}$ might reach a state in $S^{\emptyset}_{\mathcal{A}}$. A state outside $S^{pe}_{\mathcal{A}}$ cannot reach $S^{\emptyset}_{\mathcal{A}}$.

Given an observation (ρ, t), we can compute the reach-set of $\mathcal{A}$ over (ρ, t) as $\mathcal{T}_{\mathcal{A}}(\rho, t)$, which is a singleton set, as $\mathcal{A}$ is deterministic and complete. Thus, we identify $\mathcal{T}_{\mathcal{A}}(\rho, t)$ with the unique state in it.

Now, we have $\mathcal{T}_{\mathcal{A}}(\rho, t) \in S^{pe}_{\mathcal{A}}$ if and only if there exists $\rho' \in T\Sigma^*$ such that for all $\mu \in T\Sigma^\omega$ we have $\rho \cdot_t \rho' \cdot \mu \notin L(\mathcal{A})$. If the latter condition is violated, then monitoring any extension of (ρ, t) will not provide the verdict $\bot$, i.e., $V_{L(\mathcal{A})}(\rho' \cdot_t \rho') \neq \bot$ for all $\rho' \in T\Sigma^*$.

Since DTMA are complementable by changing the acceptance condition, but not the set of locations and clocks, we can compute $S^{pe}_{\overline{\mathcal{A}}}$ for the complement automaton $\overline{\mathcal{A}}$ of $\mathcal{A}$ and provide the dual characterization for the $\top$-verdict: $\mathcal{T}_{\mathcal{A}}(\rho, t) \in S^{pe}_{\overline{\mathcal{A}}}$ if and only if there exists $\rho' \in T\Sigma^*$ such that for all $\mu \in T\Sigma^\omega$ we have $\rho \cdot_t \rho' \cdot \mu \in L(\mathcal{A})$. Again, if the latter condition is violated, then monitoring any extension of (ρ, t) will not provide the verdict $\top$, i.e., $V_{L(\mathcal{A})}(\rho' \cdot_t \rho') \neq \top$ for all $\rho' \in T\Sigma^*$.

With these characterizations, we can conclude whether the $\top$ or $\bot$ verdict are still possible for some extensions, which allows us to decide weak (ρ, t)-monitorability: $L(\mathcal{A})$ is weakly (ρ, t)-monitorable if and only if $\mathcal{T}_{\mathcal{A}}(\rho, t) \in S^{pe}_{\mathcal{A}} \cup S^{pe}_{\overline{\mathcal{A}}}$. Note that this characterization crucially depends on the fact that both $\mathcal{A}$ and $\overline{\mathcal{A}}$ have the same states (locations and clock valuations).

For strong (ρ, t)-monitorability, we need to compute the states that can leave the possibly empty states via a finite run: $\mathcal{A}$ is strongly (ρ, t)-monitorable if and only if $\mathcal{T}_{\mathcal{A}}(\rho, t) \notin Pre^*_{\mathcal{A}}(\overline{S^{pe}_{\mathcal{A}}}) \cap Pre^*_{\mathcal{A}}(\overline{S^{pe}_{\overline{\mathcal{A}}}})$.

Due to Lemma 17 and Lemma 18, both characterizations are effectively decidable. ◀

Due to Remark 12, we also obtain the decidability of strong and weak monitorability.

▶ **Corollary 20.** *Strong and weak monitorability are decidable for properties given by DTMA.*

5 Monitorability with Step-Bounded Horizons

Strong monitorability of a property ensures that at any time during monitoring, the observation (ρ, t) made so far can be extended by some finite ρ' such that a conclusive verdict can be made after $\rho \cdot_t \rho'$. Thus, in the setting of strong monitorability, it is always meaningful to keep monitoring. In contrast, for weakly monitorable properties, the ability to make a conclusive verdict after some future (finite) monitoring is not guaranteed. Ideally, we would like to refine the rather uninformative verdict *?* with guaranteed minimum bounds on the number of future events before a positive or negative verdict can be made. In case both these bounds are ∞, further monitoring is useless as the verdicts reported will always be *?*. Here it is prudent to distinguish between the two definitive verdicts, as they have different decidability properties.

▶ **Definition 21** (Weak Monitorability with Step-Bounded Horizons). *Fix a property $\varphi \subseteq T\Sigma^\omega$, an observation (ρ, t), and a bound $n \in \mathbb{N}$.*

- *φ is bounded weakly $\top$-(ρ, t)-monitorable with respect to n if and only if*

  there exists $\rho' \in T\Sigma^{\leq n}$ such that $V_\varphi(\rho \cdot_t \rho') = \top$.

- *φ is bounded weakly $\bot$-(ρ, t)-monitorable with respect to n if and only if*

  there exists $\rho' \in T\Sigma^{\leq n}$ such that $V_\varphi(\rho \cdot_t \rho') = \bot$.

- *φ is bounded weakly (ρ, t)-monitorable with respect to n if and only if*

  there exists $\rho' \in T\Sigma^{\leq n}$ such that $V_\varphi(\rho \cdot_t \rho') \in \{\top, \bot\}$.

- *φ is bounded weakly $\top$-monitorable with respect to n if it is bounded weakly $\top$-$(\varepsilon, 0)$-monitorable with respect to n.*
- *φ is bounded weakly $\bot$-monitorable with respect to n if it is bounded weakly $\bot$-$(\varepsilon, 0)$-monitorable with respect to n.*
- *φ is bounded weakly monitorable with respect to n if it is bounded weakly $(\varepsilon, 0)$-monitorable with respect to n.*

In the following, we show that bounded weak monitorability and bounded weak $\top$-monitorability are undecidable, but that bounded weak $\bot$-monitorability is decidable.

▶ **Theorem 22.** *Bounded weak $\top$-monitorability is undecidable for properties given by TBA.*

Proof. Let φ be a property given by a TBA $\mathcal{A}$. Then universality of $\mathcal{A}$ is equivalent to bounded weak $\top$-monitorability of φ for the bound $n = 0$. As universality for timed automata is undecidable [3], 0-bounded weak $\top$-monitorability for TBA properties is undecidable. ◀

Next, we show that weak $\bot$-(ρ, t)-monitorability behaves differently.

▶ **Theorem 23.** *Bounded weak $\bot$-(ρ, t)-monitorability is decidable for properties given by TBA.*

Proof. Let (ρ, t) be an observation with $\rho = (\sigma_1, t_1)(\sigma_2, t_2)\cdots(\sigma_m, t_m)$ and let the property be given by the TBA $\mathcal{A}$.

Let $\pi = q_0 \xrightarrow{\sigma_1,\lambda_1,g_1} q_1 \cdots q_{m+n-1} \xrightarrow{\sigma_{m+n},\lambda_{m+n},g_{m+n}} q_{m+n}$ be a syntactic path of length $m+n$ of the TBA $\mathcal{A}$ induced by transitions $(q_i, q_{i+1}, \sigma_i, \lambda_i, g_i)$. Also, let $\tau = \{\tau_1, \ldots, \tau_{m+n}\}$ be the set of variables ranging over global time (i.e., $\mathbb{R}_{\geq 0}$), where τ_i denotes the time at which the i-th transition is taken. Clearly, $\tau_i = t_i$ for $i = 1, \ldots, m$. Most importantly, there is a zone $Z_\pi(\tau)$ over τ that precisely captures when (σ, τ) is a timed word realizing π. In this case, we may also express the resulting clock valuation after realizing (σ, τ) on π as $v_\pi(\tau) = (v_\pi^1, \ldots, v_\pi^k)$, where k is the number of clocks of $\mathcal{A}$ and $v_\pi^i = \tau_{m+n} - \tau_{\ell(i)}$, with $\ell(i)$ being the index of the last transition when the clock x_i was reset. Now, if $(q_{m+n}, v_\pi(\tau)) \notin S_{\mathcal{A}}^{ne}$, then the run of (σ, τ) following π cannot be extended to an accepting run of $\mathcal{A}$.

Witnesses for weak monitorability are closed under extensions. Hence, in the following construction, we can restrict ourselves w.l.o.g. to words of length exactly n instead of words of length at most n. The following formula expresses the existence of a timed word $\rho' = (\sigma', \tau')$ of length n such that all runs on $\rho \cdot_t \rho'$ end in a state outside of $S_{\mathcal{A}}^{ne}$, witnessing $V_{L(\mathcal{A})}(\rho \cdot_t \rho') = \bot$:

$$\exists \tau_1 \ldots \exists \tau_{m+n}.\ \bigwedge_{i=1}^{m} \tau_i = t_i \wedge \tau_m \leq t \leq \tau_{m+1} \wedge \left(\bigvee_{\sigma_{m+1}\cdots\sigma_{m+n} \in \Sigma^n} \bigwedge_{\pi \in \mathrm{SP}} (Z_\pi(\tau) \rightarrow v_\pi(\tau) \notin S_{\mathcal{A}}^{ne}) \right),$$

where SP is the set of syntactic paths of the form

$$\pi = q_0 \xrightarrow{g_1,\sigma_1,\lambda_1} q_1 \cdots \xrightarrow{g_m,\sigma_m,\lambda_m} q_m \cdots q_{m+n-1} \xrightarrow{\sigma_{m+n},\lambda_{m+n},g_{m+n}} q_{m+n}.$$

The formula above expresses the existence of values for $\tau_1, \ldots, \tau_{m+n}$, with $\tau_i = t_i$ for $i = 1, \ldots, m$, and $\tau_m \leq t \leq \tau_{m+1}$, ensuring that only extensions of the observation (ρ, t) are considered. The last part of the formula ensures the existence of a timed word $\rho' = (\sigma', \tau')$, where $\sigma' = \sigma_{m+1} \cdots \sigma_{m+n}$ and $\tau' = \tau_{m+1} \cdots \tau_{m+n}$, where all runs – which are captured by $Z_\pi(\tau)$ – are outside $S_{\mathcal{A}}^{ne}$. The formula is in the first order theory of real-closed fields, which is decidable [33]. Thus, bounded weak $\bot$-monitorability is decidable for properties given by TBA. ◀

▶ Remark 24. It follows from Theorem 23 that for properties where both the property and its complement are given by TBA, bounded weak monitorability is decidable. Hence, if the property is given by a DTMA, then bounded weak monitorability is decidable. Moreover, one can even compute the tightest bound n by doing a breadth-first search over the symbolic state graph of the deterministic timed automaton, searching for a (shortest) path to the empty language states.

Lastly, we prove bounded weak monitorability undecidable for properties given by TBA.

▶ **Theorem 25.** *Bounded weak monitorability is undecidable for properties given by TBA.*

Proof. In the second step of the undecidability proof of weak monitorability (see Theorem 14), we have shown how to reduce weak $\top$-monitorability to weak monitorability. The same construction reduces bounded weak $\top$-monitorability to bounded weak monitorability, as the length of the witnesses is only decreased by removing #'s. ◀

Due to Remark 12, we also obtain results for the remaining cases.

▶ **Corollary 26.** *Bounded weak $\top$-(ρ,t)-monitorability and bounded weak (ρ,t)-monitorability are undecidable (even for fixed (ρ,t)) while bounded weak $\bot$-monitorability is decidable.*

6 Refined Monitoring with Time-Horizon Verdicts

In the previous section, we have shown that the uninformative verdict *?* can be refined by checking whether within a bounded number of new observations, a definitive verdict may be given. In this section, we again refine the uninformative verdict *?* by computing lower bounds on the time that needs to pass (independently of the number of events observed during that time) before a definitive verdict may be given. Again, if this is infinite, then the monitoring process can be stopped, as no amount of waiting will yield a definitive verdict. This refinement was introduced and briefly studied as "time predictive monitoring" by Grosen et al. [23]. Here we revisit (and rename) this notion by proving computability of the time-bounded monitorability for properties given by a DTMA.

Recall that time passing without any new observed events can nevertheless yield definitive verdicts (see, e.g., the last two items in Example 7). Thus, in a practical setting, one is interested in intermittently querying the monitoring function even if no events are observed. Here, we give lower bounds on the time one should let pass before the next such query is made, thereby reducing the computational overhead of these queries.

▶ **Definition 27** (Refined Monitoring with Time-Horizon Verdicts). *Given an observation (ρ,t) and a property $\varphi \subseteq T\Sigma^\omega$, the refined monitoring function P_φ is defined as*

$$P_\varphi(\rho,t) = \begin{cases} \top & \text{if } V_\varphi(\rho,t) = \top, \\ \bot & \text{if } V_\varphi(\rho,t) = \bot, \\ (P_\varphi^\top(\rho,t), P_\varphi^\bot(\rho,t)) & \text{otherwise,} \end{cases}$$

with

- $P_\varphi^\top(\rho,t) = \inf\{t' \mid \rho' \in T\Sigma^* \text{ such that } V_\varphi(\rho \cdot_t \rho', t') = \top \text{ and } t' \geq \tau(\rho \cdot_t \rho')\}$ *and*
- $P_\varphi^\bot(\rho,t) = \inf\{t' \mid \rho' \in T\Sigma^* \text{ such that } V_\varphi(\rho \cdot_t \rho', t') = \bot \text{ and } t' \geq \tau(\rho \cdot_t \rho')\}$,

where we use the convention $\inf \emptyset = \infty$.

▶ **Example 28.** Consider the MITL property $\varphi = F_{[20,40]}b$. Monitoring the finite timed word $\rho = (a,5.1)(c,21.0)(c,30.4)(b,35.1)(a,40.2)$ will result in three *?* verdicts followed by the verdict $\top$ when $(b,35.1)$ is read. However, we may offer significantly more information, e.g., when reading $(a,5.1)$ it is clear that at least 14.9 time-units must elapse before we can give the verdict $\top$, and at least 34.9 time-units must elapse before we can give the verdict $\bot$. Hence, $P_\varphi((a,5.1),5.1) = (14.9, 34.9)$.

▶ Remark 29. $V_\varphi(\rho,t) = \top$ implies $P_\varphi^\top(\rho,t) = 0$ and $P_\varphi^\bot(\rho,t) = \infty$ and $V_\varphi(\rho,t) = \bot$ implies $P_\varphi^\top(\rho,t) = \infty$ and $P_\varphi^\bot(\rho,t) = 0$, but the converse is in general not true. Consider, for example, the MITL property $\varphi = F_{\geq 0}a$ and the observation $\rho = (b,1)$. Then, $P_\varphi^\top(\rho,1) = 0$ (witnessed by $\rho' = (a,0)$ for which we have $V_{\rho \cdot_t \rho'} = \top$) and $P_\varphi^\bot = \infty$, but $V_\varphi(\rho,1) =$ *?*.

Time-bounded monitorability for an observation (ρ, t) refines weak (ρ, t)-monitorability.

▶ **Lemma 30.** *Let (ρ, t) be an observation and φ a property. Then, φ is weakly (ρ, t)-monitorable if and only if at least one of the values $P^{\top}_{\varphi}(\rho, t)$ and $P^{\bot}_{\varphi}(\rho, t)$ is finite.*

Proof. Let φ be weakly (ρ, t)-monitorable, i.e., there is a $\rho' \in T\Sigma^*$ such that $V_\varphi(\rho \cdot_t \rho') \in \{\top, \bot\}$, say it is $\top$ (the other case is analogous). Then, ρ' witnesses $P^{\top}_{\varphi}(\rho, t) \leq \tau(\rho')$, i.e., $P^{\top}_{\varphi}(\rho, t)$ is finite. On the other hand, if (say) $P^{\top}_{\varphi}(\rho, t)$ is finite (the other case is analogous), then there is a ρ' such that $V_\varphi(\rho \cdot_t \rho') = \top$. Hence, φ is weakly (ρ, t)-monitorable. ◀

▶ **Theorem 31** (Refined Monitoring is Effective)**.** *P_φ is effectively computable, if φ is given by a DTMA.*

Proof. When online-monitoring φ over some observation (ρ, t), we compute the reach-set $\mathcal{T}_{\mathcal{A}}(\rho, t)$ and check if it is a subset of the non-empty language states $S^{ne}_{\mathcal{A}}$ or a subset of the empty language states $S^{\emptyset}_{\mathcal{A}}$ [23]. Thus, $P^{\bot}_{\varphi}(\rho, t)$ is the infimum of the time duration of all paths from $\mathcal{T}_{\mathcal{A}}(\rho, t)$ to $S^{\emptyset}_{\mathcal{A}}$. This is a time-optimal reachability problem.

Asarin and Maler [7] introduced the concept of time-optimal strategies for timed game automata. Since timed game automata trivially generalize timed automata, we can adapt the computation of the optimal time bound to obtain the minimal possible time to reach an empty language state from the reach-set. This gives us the value $P^{\bot}_{\mathcal{A}}(\rho, t)$.

Since DTMA are closed under complement, the same procedure for the complement automaton gives us the value $P^{\top}_{\mathcal{A}}(\rho, t)$. ◀

As explained in the proof of Theorem 31, monitoring is implemented by keeping track of the reach-set of the observation and checking at each update whether it is contained in the non-empty language states or in the empty language states. But, as explained in the introduction of this section, one should not only update the reach-set when a new event is observed, but also intermittently when time has passed.

For this special case where only time passes, which is covered in Definition 27 by considering $\rho' = \varepsilon$ in the infimum, we do not need to solve the expensive time-optimal reachability problem described in the proof of Theorem 31. Instead, we rely on zone operations to compute $\delta_\varphi(\rho, t) = \inf\{d \mid V_\varphi(\rho, t + d) \in \{\top, \bot\}\}$. This is done by exploring all delays (removing the upper bounds of the zones) and subtracting the non-empty language states. The lower bounds in the zones of the resulting states gives the minimum time a verdict can be made by waiting. Now, having observed ρ and the current time being t such that $V_\varphi(\rho, t) = ?$, querying the monitoring function again without a new observation before time $t + \delta_\varphi(\rho, t)$ will not yield a different verdict and can thus be avoided.

7 Related Work

A formal notion of monitorability was first introduced by Pnueli and Zaks in their work on monitoring Property Specification Language (PSL) [29]. In that work, the authors defined strong monitorability given a finite prefix, called σ-monitorability, on which we base our (ρ, t)-monitorability. From this, Bauer, Leuker, and Schallhart defined the most common definition of monitorability, that is strong σ-monitorability from the initial state [10]. They proved that safety and guarantee properties are a proper subset of the class of strongly monitorable properties. Later, Chen et al. [13] and Peled and Havelund [28] noticed that there exist properties that are not strongly monitorable but still have utility to monitor, and proposed equivalent definitions of weak monitorability. Mascle et al. showed the monitorability of

LTL properties can be improved by considering robust semantics [27]. The term *strong monitorability* has been used before in the context of partially observable stochastic systems modeled as Hidden Markov Models. Sistla, Žefran, and Feng first used the term, contrasted with (standard) monitorability [32] in their work extending the results of Gondi, Patel, and Sistla on monitoring ω-regular properties of stochastic systems [21].

The complexity of monitorability problems has been studied in other untimed settings. Diekert and Leuker proposed a topological definition of strong monitorability, showing that problem is equivalent to showing that the boundary in the Cantor topology has an empty interior [16]. Diekert, Muscholl, and Walukiewicz later proved that deciding monitorability for (untimed) Büchi automata is PSPACE-complete [18]. Agrawal and Bonakdarpour proposed a definition of monitorability for hyperproperties and determined the monitorable classes for their three-valued specification language, HyperLTL [15]. Francalanza, Aceto, and Ingolfsdottir characterized monitorable properties of the branching-time μ-Hennessy-Milner Logic [19]. This was later extended by Aceto et al. who introduced a hierarchy of monitorable fragments of the language [1].

Attempts have been made to unify these different notions of monitorability. Peled and Havelund introduced a classification for properties centered around monitorability [28]. Kauffman, Havelund, and Fischmeister defined a common notation for strong and weak monitorability for different verdict domains [24]. Aceto, Achilleos, and Francalanza provided syntactic characterizations of monitorability for classical notions of monitorability as well as for a variant of the modal μ-calculus, recHML [2].

We are aware of only one other work addressing monitorability for real-time properties. Amara et al. (very recently) introduced a new linear-time timed μ-calculus, which subsumes MTL, and therefore also MITL, and identified its largest monitorable fragment [6]. Their work differs from ours in three important respects. First, they rely on a definition of monitorability introduced by Schneider as *Execution Monitoring Enforceability* [31] while we use the more typical definition of strong and weak monitorability due to Pnueli and Zaks [29]. Second, they introduce a new calculus and characterize its maximal monitorable fragment. We instead consider the case of properties expressed as Timed Automata, which lend themselves to algorithmic manipulation. Finally, we prove decidability and undecidability results.

8 Conclusion

In this work, we have studied monitorability for timed properties specified by either (possibly nondeterministic) TBA or by deterministic TMA. In general, we proved monitorability decidable for specifications given by deterministic automata and undecidable for specifications given by nondeterministic automata. The notable exception here is bounded weak $\bot$-monitorability, which is even decidable for nondeterministic TBA.

Also, we provided refinements of monitoring and monitorability making the verdict *?* more informative by providing bounds on the number of events or the amount of time that needs to pass before a conclusive verdict may be given. In practical settings, this is crucial information and also allows to optimize the monitoring process in the real-time setting.

Our decidability proof for monitorability of DTMA relies on the fact that DTMA can be complemented without changing the state space. On the other hand, monitorability is undecidable if the property is given by a TBA. Thus, another question for further research is to consider strong monitorability when given TBA for the property *and* for its negation. This is a very natural setting, as the specification logic MITL is closed under negation and can be translated into TBA. Also, monitoring often requires TBA for both the property and its negation [23, 20, 14].

References

1 Luca Aceto, Antonis Achilleos, Adrian Francalanza, Anna Ingólfsdóttir, and Karoliina Lehtinen. Adventures in monitorability: From branching to linear time and back again. In *Symposium on Principles of Programming Languages (POPL'19)*, volume 3. ACM, January 2019. doi:10.1145/3290365.

2 Luca Aceto, Antonis Achilleos, Adrian Francalanza, Anna Ingólfsdóttir, and Karoliina Lehtinen. An operational guide to monitorability with applications to regular properties. *Software and Systems Modeling*, 20(2):335–361, April 2021. doi:10.1007/s10270-020-00860-z.

3 Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994. doi:10.1016/0304-3975(94)90010-8.

4 Rajeev Alur, Tomás Feder, and Thomas A. Henzinger. The benefits of relaxing punctuality. *J. ACM*, 43(1):116–146, 1996. doi:10.1145/227595.227602.

5 Rajeev Alur and P. Madhusudan. Decision problems for timed automata: A survey. In *Formal Methods for the Design of Real-Time Systems: International School on Formal Methods for the Design of Computer, Communication, and Software Systems*, volume 3185 of *LNCS*, pages 1–24. Springer, 2004. doi:10.1007/978-3-540-30080-9_1.

6 Mouloud Amara, Giovanni Bernardi, Mohammed Foughali, and Adrian Francalanza. A theory of (linear-time) timed monitors. In *39th European Conference on Object-Oriented Programming (ECOOP 2025)*, volume 333, Bergen (NO), Norway, June 2025. URL: https://hal.science/hal-05043055.

7 Eugene Asarin and Oded Maler. As soon as possible: Time optimal control for timed automata. In Frits W. Vaandrager and Jan H. van Schuppen, editors, *Hybrid Systems: Computation and Control, Second International Workshop, HSCC'99, Berg en Dal, The Netherlands, March 29-31, 1999, Proceedings*, volume 1569 of *LNCS*, pages 19–30. Springer, 1999. doi:10.1007/3-540-48983-5_6.

8 David Basin, Felix Klaedtke, and Eugen Zălinescu. Algorithms for monitoring real-time properties. In *Runtime Verification*, pages 260–275. Springer, 2012. doi:10.1007/978-3-642-29860-8_20.

9 Andreas Bauer, Martin Leucker, and Christian Schallhart. Monitoring of real-time properties. In *Foundations of Software Technology and Theoretical Computer Science*, pages 260–272, Berlin, Heidelberg, 2006. Springer. doi:10.1007/11944836_25.

10 Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for LTL and TLTL. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(4):14:1–14:64, September 2011. doi:10.1145/2000799.2000800.

11 Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In Jörg Desel, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Lectures on Concurrency and Petri Nets, Advances in Petri Nets [This tutorial volume originates from the 4th Advanced Course on Petri Nets, ACPN 2003, held in Eichstätt, Germany in September 2003. In addition to lectures given at ACPN 2003, additional chapters have been commissioned]*, volume 3098 of *Lecture Notes in Computer Science*, pages 87–124. Springer, 2003. doi:10.1007/978-3-540-27755-2_3.

12 Thomas Brihaye, Gilles Geeraerts, Hsi-Ming Ho, and Benjamin Monmege. MightyL: A compositional translation from MITL to timed automata. In Rupak Majumdar and Viktor Kuncak, editors, *CAV 2017, Part I*, volume 10426 of *LNCS*, pages 421–440, Cham, 2017. Springer. doi:10.1007/978-3-319-63387-9_21.

13 Zhe Chen, Yifan Wu, Ou Wei, and Bin Sheng. Deciding weak monitorability for runtime verification. In *Int. Conference on Software Engineering (ICSE'18)*, pages 163–164. ACM, 2018. doi:10.1145/3183440.3195077.

14 Alessandro Cimatti, Thomas Møller Grosen, Kim G. Larsen, Stefano Tonetta, and Martin Zimmermann. Exploiting assumptions for effective monitoring of real-time properties under partial observability. In Alexandre Madeira and Alexander Knapp, editors, *SEFM*, volume 15280 of *LNCS*, pages 70–88. Springer, 2024. doi:10.1007/978-3-031-77382-2_5.

15 Michael R. Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K. Micinski, Markus N. Rabe, and César Sánchez. Temporal logics for hyperproperties. In *Int. Conference on Principles of Security and Trust (POST'14)*, volume 8414 of *LNCS*, pages 265–284. Springer, 2014. doi:10.1007/978-3-642-54792-8_15.

16 Volker Diekert and Martin Leucker. Topology, monitorable properties and runtime verification. *Theoretical Computer Science*, 537:29–41, 2014. doi:10.1016/j.tcs.2014.02.052.

17 Volker Diekert and Anca Muscholl. On distributed monitoring of asynchronous systems. In *Logic, Language, Information and Computation*, volume 7456 of *LNTCS*, pages 70–84. Springer, 2012. doi:10.1007/978-3-642-32621-9_5.

18 Volker Diekert, Anca Muscholl, and Igor Walukiewicz. A note on monitors and Büchi automata. In *Theoretical Aspects of Computing - ICTAC 2015*, volume 9399 of *LNTCS*, pages 39–57. Springer, 2015. doi:10.1007/978-3-319-25150-9_3.

19 Adrian Francalanza, Luca Aceto, and Anna Ingolfsdottir. Monitorability for the Hennessy-Milner logic with recursion. *Formal Methods in System Design*, 51(1):87–116, August 2017. doi:10.1007/s10703-017-0273-z.

20 Martin Fränzle, Thomas Møller Grosen, Kim G. Larsen, and Martin Zimmermann. Monitoring real-time systems under parametric delay. In Nikolai Kosmatov and Laura Kovács, editors, *IFM 2024*, volume 15234 of *LNCS*, pages 194–213. Springer, 2024. doi:10.1007/978-3-031-76554-4_11.

21 Kalpana Gondi, Yogeshkumar Patel, and A. Prasad Sistla. Monitoring the full range of ω-regular properties of stochastic systems. In Neil D. Jones and Markus Müller-Olm, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 105–119, Berlin, Heidelberg, 2009. Springer. doi:10.1007/978-3-540-93900-9_12.

22 Thomas Møller Grosen, Sean Kauffman, Kim G. Larsen, and Martin Zimmermann. Time for timed monitorability. *arXiv*, 2504.10008, 2025. doi:10.48550/arXiv.2504.10008.

23 Thomas Møller Grosen, Sean Kauffman, Kim Guldstrand Larsen, and Martin Zimmermann. Monitoring timed properties (revisited). In Sergiy Bogomolov and David Parker, editors, *FORMATS 2022*, volume 13465 of *LNCS*, pages 43–62. Springer, 2022. doi:10.1007/978-3-031-15839-1_3.

24 Sean Kauffman, Klaus Havelund, and Sebastian Fischmeister. What can we monitor over unreliable channels? *International Journal on Software Tools for Technology Transfer*, 23:579–600, June 2021. doi:10.1007/s10009-021-00625-z.

25 Kim Guldstrand Larsen, Marius Mikucionis, and Brian Nielsen. Online testing of real-time systems using uppaal. In *Formal Approaches to Software Testing*, volume 3395 of *LNCS*, pages 79–94. Springer, 2004. doi:10.1007/978-3-540-31848-4_6.

26 N.G. Leveson and C.S. Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, 1993. doi:10.1109/MC.1993.274940.

27 Corto Mascle, Daniel Neider, Maximilian Schwenger, Paulo Tabuada, Alexander Weinert, and Martin Zimmermann. From LTL to rltl monitoring: improved monitorability through robust semantics. *Formal Methods Syst. Des.*, 59(1):170–204, 2021. doi:10.1007/S10703-022-00398-4.

28 Doron Peled and Klaus Havelund. Refining the safety–liveness classification of temporal properties according to monitorability. In *Models, Mindsets, Meta: The What, the How, and the Why Not? Essays Dedicated to Bernhard Steffen on the Occasion of His 60th Birthday*, volume 11200 of *LNCS*, pages 218–234. Springer, 2019. doi:10.1007/978-3-030-22348-9_14.

29 A. Pnueli and A. Zaks. PSL model checking and run-time verification via testers. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006: Formal Methods*, pages 573–586, Berlin, Heidelberg, 2006. Springer. doi:10.1007/11813040_38.

30 Narad Rampersad, Jeffrey O. Shallit, and Zhi Xu. The computational complexity of universality problems for prefixes, suffixes, factors, and subwords of regular languages. *Fundam. Informaticae*, 116(1-4):223–236, 2012. doi:10.3233/FI-2012-680.

31 Fred B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, February 2000. doi:10.1145/353323.353382.

32 A. Prasad Sistla, Miloš Žefran, and Yao Feng. Monitorability of stochastic dynamical systems. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, pages 720–736, Berlin, Heidelberg, 2011. Springer. doi:10.1007/978-3-642-22110-1_58.

33 Alfred Tarski. A decision method for elementary algebra and geometry. In Bob F. Caviness and Jeremy R. Johnson, editors, *Quantifier Elimination and Cylindrical Algebraic Decomposition*, pages 24–84, Vienna, 1998. Springer Vienna.

34 Prasanna Thati and Grigore Roşu. Monitoring algorithms for metric temporal logic specifications. *Electronic Notes in Theoretical Computer Science*, 113:145–162, 2005. doi:10.1016/j.entcs.2004.01.029.

Compositional Active Learning of Synchronizing Systems Through Automated Alphabet Refinement

Léo Henry
Royal Holloway, University of London, UK

Mohammad Reza Mousavi
King's College London, UK

Thomas Neele
Eindhoven University of Technology, The Netherlands

Matteo Sammartino
Royal Holloway, University of London, UK

Abstract

Active automata learning infers automaton models of systems from behavioral observations, a technique successfully applied to a wide range of domains. Compositional approaches for concurrent systems have recently emerged. We take a significant step beyond available results, including those by the authors, and develop a general technique for compositional learning of a synchronizing parallel system with an unknown decomposition. Our approach *automatically refines* the global alphabet into component alphabets while learning the component models. We develop a theoretical treatment of *distributions* of alphabets, i.e., sets of possibly overlapping component alphabets. We characterize *counter-examples* that reveal inconsistencies with global observations, and show how to systematically update the distribution to restore consistency. We present a compositional learning algorithm implementing these ideas, where learning counterexamples precisely correspond to distribution counterexamples under well-defined conditions. We provide an implementation, called CoalA, using the state-of-the-art active learning library LearnLib. Our experiments show that in more than 630 subject systems, CoalA delivers orders of magnitude improvements (up to five orders) in membership queries and in systems with significant concurrency, it also achieves better scalability in the number of equivalence queries.

2012 ACM Subject Classification Theory of computation → Active learning

Keywords and phrases Active learning, Compositional methods, Concurrency theory, Labelled transition systems, Formal methods

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2025.20

Supplementary Material *Software (Source Code)*: `https://doi.org/10.5281/zenodo.15170685`

Funding *Léo Henry*: EPSRC project Verification of Hardware Concurrency via Model Learning (CLeVer) – EP/S028641/1.
Mohammad Reza Mousavi: UKRI Trustworthy Autonomous Systems Node in Verifiability – EP/V026801/2; EPSRC project Verified Simulation for Large Quantum Systems (VSL-Q) – EP/Y005244/1; EPSRC project Robust and Reliable Quantum Computing (RoaRQ), Investigation 009 Model-based monitoring and calibration of quantum computations (ModeMCQ) – EP/W032635/1; ITEA/InnovateUK projects GENIUS (600642) and GreenCode (600643).
Thomas Neele: NWO grant VI.Veni.232.224.
Matteo Sammartino: EPSRC project Verification of Hardware Concurrency via Model Learning (CLeVer) – EP/S028641/1.

Acknowledgements We thank the reviewers for their thorough comments and suggestions, and the authors of [27] for their feedback on an earlier version of this paper.

© Léo Henry, Mohammad Reza Mousavi, Thomas Neele, and Matteo Sammartino;
licensed under Creative Commons License CC-BY 4.0

1 Introduction

Automata learning [16] has been successfully applied to learn widely-used protocols such as TCP [12], SSH [13], and QUIC [11], CPU caching policies [38] and finding faults in their black-box actual implementations. Moreover it has been applied to a wide range of applications such as bank cards [1] and biometric passports [2, 28]. There are already accessible expositions of the success stories in this field [20, 36]. However, it is well known that the state-of-the-art automata learning algorithms do not scale beyond systems with more than a few hundreds of input and output symbols in their alphabets [36].

Scalability to larger systems with large input alphabets is required for many real-life systems. This requirement has inspired some recent attempts [25, 31, 27] to come up with compositional approaches to automata learning, to address the scalability issues. Some of the past compositional approaches [31] relied on an a-priori knowledge of the system decomposition, while others [25] considered non-synchronizing state machines or a subset of synchronizing ones to help learning the decomposition [27]. These assumptions need to be relaxed to enable dealing with legacy and black box systems and dealing with synchronizing components. In particular, in the presence of an ever-increasing body of legacy automatically-generated code, architectural discovery is a significant challenge [23, 15, 32]. This is a significant challenge, because refactoring and rejuvenating legacy systems has been posed as a significant application of automata learning [33].

In this paper, we take a significant step beyond the available results and develop a compositional automata learning approach that does not assume any pre-knowledge of the decomposition of the alphabet and allows for an arbitrary general synchronization scheme, common in the theory of automata and process calculi [19]. To this end, we take inspirations from the realizability problem in concurrency theory and use iterative refinements of the alphabet decomposition (called distributions [30]) to arrive in a provably sound decomposition while learning the components' behavior. To our knowledge this is the first result of its kind and the first extension of realizability into the domain of automata learning.

To summarize, the contributions of our paper are listed below:

- We develop a novel theory of system decomposition for LTS synchronization that formally characterizes which alphabet decompositions can accurately model observed behaviors, establishing a theoretical foundation for automated component discovery. Proofs of our theorems are given in the appendix.
- Based on this, we propose a compositional active learning algorithm that dynamically refines component alphabets during the learning process, supporting standard synchronization mechanisms without requiring a priori knowledge of the system's decomposition.
- We implemented our approach as the prototype tool CoalA, built on the state-of-the-art LearnLib framework [22], and evaluated it on over 630 systems from three benchmark sets. Compared to a monolithic approach, CoalA achieved substantial reductions in queries, with up to five orders of magnitude fewer membership queries and one order fewer equivalence queries across most of our benchmark systems with parallel components, resulting in better overall scalability. The replication package is available at [18].

2 Related work

Realizability of sequential specifications in terms of parallel components has been a long-standing problem in concurrency theory. In the context of Petri nets, this has been pioneered by Ehrenfeucht and Rozenberg [10], followed up by the work of Castellani, Mukund and Thiagarajan [8]. Realizability has been further investigated in other models of concurrency

such as team automata [34], session types [5], communicating automata [17] and labelled transition systems (LTSs) [35]. Related to this line of research is the decomposition of LTSs into prime processes [26]. We are inspired by the work of Mukund [30], characterizing the transition systems that can be synthesized into an equivalent parallel system given a decomposition of their alphabet (called distribution). Mukund explores this characterization for two notions of parallel composition (loosely cooperating- and synchronous parallel composition) and three notions of equivalence (isomorphism, language equivalence, and bisimulation). We base our work on the results of Mukund for loosely cooperating systems and language equivalence. We extend it to define consistency between observations and distributions and refining distributions to reinstate consistency.

Our work integrates two recent approaches on compositional learning: we extend the work on learning synchronous parallel composition of automata [31] by automatically learning the decomposition of the alphabets, through refinement of distributions; moreover, we extend the work on learning interleaving parallel composition of automata [25] by enabling a generic synchronization scheme among components. In parallel to our work, an alternative proposal [27] has appeared to allow for synchronization among interleaving automata; however, the proposed synchronization scheme assumes that whenever two components are not ready to synchronize, e.g., because they produce different outputs on the same input, a special output is produced (or otherwise, the semantic model is output deterministic). We do not assume any such additional information and use a synchronization scheme widely used in the theory of automata and process calculi [19]. Other contributions related to compositional learning include the active learning of *product automata*, a variation of Mealy machine where the output is the combination of outputs of several Mealy machines – as in our case, the component Mealy machines are learned individually [29]; learning of *systems of procedural automata* [14], sets of automata that can call each other in a way similar to procedure calls; learning asynchronously-communicating finite state machines via queries in the form of message sequence charts [6], though using a monolithic approach.

3 Preliminaries

We use Σ to denote a *finite alphabet* of action symbols, and $\Sigma^\star$ to denote the set of finite sequences of symbols in Σ, which we call *traces*; we use $\epsilon \in \Sigma^\star$ to denote the empty trace. Given two traces $\sigma_1, \sigma_2 \in \Sigma^\star$, we denote their concatenation by $\sigma_1 \cdot \sigma_2$. We refer to the ith element of σ by $\sigma[i]$. The *projection* $\sigma_{\restriction \Sigma'}$ of σ on an alphabet $\Sigma' \subseteq \Sigma$ is the sequence of symbols in σ that are also contained in Σ': $\epsilon_{\restriction \Sigma'} = \epsilon$ and $\sigma \cdot a_{\restriction \Sigma'} = \sigma_{\restriction \Sigma'} \cdot a$ if $a \in \Sigma'$ and $\sigma_{\restriction \Sigma'}$ otherwise. We generalize this notation to sets (and thus languages), such that $S_{\restriction \Sigma'} = \{\sigma_{\restriction \Sigma'} \mid \sigma \in S\}$. Given a set S, we write $|S|$ for its cardinality. We write $\mathsf{img}(f)$ for the image of a function f.

3.1 Labelled Transition Systems

In this work we represent the state-based behavior of a system as a *labelled transition system*.

▶ **Definition 3.1** (Labelled Transition System). *A* labelled transition system *(LTS) is a four-tuple $T = (S, \Sigma, \to, \hat{s})$, where S is a set of states; Σ is a finite alphabet of actions; $\to \subseteq S \times \Sigma \times S$ is a transition relation; $\hat{s} \in S$ is an initial state.*

We write in infix notation $s \xrightarrow{a} t$ for $(s, a, t) \in \to$. We say that an action a is *enabled* in s, written $s \xrightarrow{a}$, if there is t such that $s \xrightarrow{a} t$. The transition relation and the notion of enabled-ness are also extended to traces $\sigma \in \Sigma^\star$, yielding $s \xrightarrow{\sigma} t$ and $s \xrightarrow{\sigma}$.

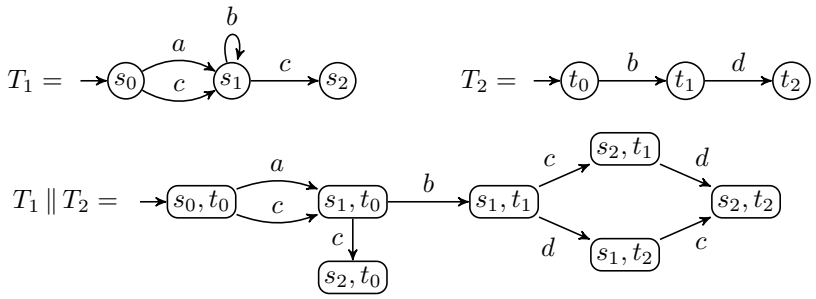


Figure 1 The parallel composition of two LTSs.

▶ **Definition 3.2** (Language of an LTS). *The language of T is the set of traces enabled from the starting state, formally:* $\mathcal{L}(T) = \{\sigma \in \Sigma^\star \mid \hat{s} \xrightarrow{\sigma}\}$.

Note that languages of LTSs are always prefix-closed, because every prefix of an enabled trace is necessarily enabled. LTSs correspond exactly to prefix-closed automata.

The parallel composition of a finite set of LTSs is a product model representing all possible behaviors when the LTSs synchronize on shared actions. Intuitively, an action a can be performed when all LTSs that have a in their alphabet can perform it in their current state. The other LTSs remain idle during the transition.

▶ **Definition 3.3** (Parallel composition). *Given n LTSs $T_i = (S_i, \Sigma_i, \to_i, \hat{s}_i)$ for $1 \leq i \leq n$, their* parallel composition*, denoted $\|_{i=1}^n T_i$, is an LTS $(S_1 \times \cdots \times S_n, \bigcup_{i=1}^n \Sigma_i, \to, (\hat{s}_1, \ldots, \hat{s}_n))$, where the transition relation $\to$ is given by the following rule:*

$$\frac{s_i \xrightarrow{a}_i t_i \quad \text{for all } i \text{ such that } a \in \Sigma_i \qquad s_j = t_j \quad \text{for all } j \text{ such that } a \notin \Sigma_j}{(s_1, \ldots, s_n) \xrightarrow{a} (t_1, \ldots, t_n)}$$

We say that an action a is *local* if there is exactly one i such that $a \in \Sigma_i$; otherwise, it is called *synchronizing*. The parallel composition of LTSs thus forces individual LTSs to cooperate on synchronizing actions; local actions can be performed independently. We typically refer to the LTSs that make up a composite LTS as *components*. For a parallel composition of two LTSs, we use the infix notation, i.e., $T \parallel T'$, when convenient. Synchronization of components corresponds to communication between components in the real world.

We define the corresponding notion for languages on restricted alphabets.

▶ **Definition 3.4** (Parallel composition of languages). *Given n languages and alphabets $(\mathcal{L}_i, \Sigma_i)$ such that $\mathcal{L}_i \subseteq \Sigma_i^\star$ for all $1 \leq i \leq n$, let $\Sigma = \bigcup_{i=1}^n \Sigma_i$. We define $\|_{i=1}^n (\mathcal{L}_i, \Sigma_i)$ as*

$$\{\sigma \in \Sigma^\star \mid \forall 1 \leq i \leq n. \sigma_{\restriction \Sigma_i} \in \mathcal{L}_i\} \ .$$

▶ **Example 3.5** (Running example). Consider the LTSs T_1 and T_2 given in Figure 1, with the respective alphabets $\{a, b, c\}$ and $\{b, d\}$. Their parallel composition is depicted at the bottom of Figure 1. Here a, c and d are local actions, whereas b is synchronizing. Note that, although T_2 can perform b from its initial state t_0, there is no b transition from (s_0, t_0) in $T_1 \parallel T_2$, because b is not enabled in s_0. Action b can only be performed in $T_1 \parallel T_2$ after T_1 does an a or a c and moves to s_1, which is captured as the a and c transitions from (s_0, t_0).

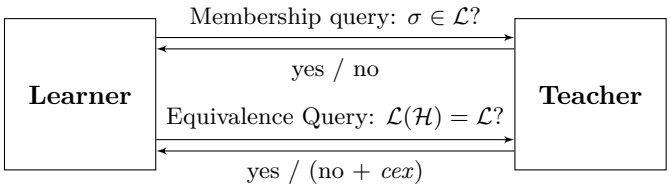


Figure 2 Active automata learning for a target language $\mathcal{L}$.

3.2 Active Automata Learning

In active automata learning [4], a *Learner* infers an automaton model of an unknown language $\mathcal{L}$ by querying a *Teacher*, which knows $\mathcal{L}$ and answers two query types (see Figure 2):

- *Membership queries*: is a trace σ in $\mathcal{L}$? The Teacher replies yes/no.
- *Equivalence queries*: given a *hypothesis* model $\mathcal{H}$, is $\mathcal{L}(\mathcal{H}) = \mathcal{L}$? The Teacher either replies yes or provides a *counter-example* – a trace that is in one language but not in the other.

Algorithms based on this framework – Angluin's $\mathtt{L}^\star$ being the classical example – converge to a canonical model (*e.g.*, the minimal DFA) of the target language. In practice, the Teacher is realized as an interface to the System Under Learning (SUL): membership queries become tests on the SUL, and equivalence queries are approximated via systematic testing strategies [7, 24].

During learning, the learner gathers *observations* about the SUL. While these observations are typically organized in a data structure (e.g., a table or a tree), they can be abstractly represented as a partial function mapping traces to their accepted (+) or rejected (−) status.

▶ **Definition 3.6** (Observation function). *An* observation function *over* Σ *is a partial function* $\mathsf{Obs} : \Sigma^\star \rightharpoonup \{+, -\}$.

We write $\mathsf{Dom}(\mathsf{Obs})$ for the domain of Obs and only consider observation functions with a finite domain. We sometimes represent an observation function Obs as the set of pairs $\{(\sigma, \mathsf{Obs}(\sigma)) \mid \sigma \in \mathsf{Dom}(\mathsf{Obs})\}$.

▶ **Definition 3.7** (Observation function/language agreement). *An observation function* Obs *agrees with a language* $\mathcal{L}$*, notation* $\mathcal{L} \models \mathsf{Obs}$*, whenever* $\sigma \in \mathcal{L} \Leftrightarrow \mathsf{Obs}(\sigma) = +$*, for all* $\sigma \in \mathsf{Dom}(\mathsf{Obs})$.

To compositionally learn a model formulated as a parallel composition of LTSs $\|_{i=1}^{n} T_i$, we define how to derive the *local* observation functions that will be used for the components.

▶ **Definition 3.8** (Local observation function). *Given a sub-alphabet* $\Sigma_i \subseteq \Sigma$*, a local observation function* $\mathsf{Obs}_{\Sigma_i} : \Sigma_i^\star \rightharpoonup \{+, -\}$ *is defined such that* $\mathsf{Dom}(\mathsf{Obs}_{\Sigma_i}) = \mathsf{Dom}(\mathsf{Obs})_{\restriction \Sigma_i}$ *and* $\mathsf{Obs}_{\Sigma_i}(\sigma') = \bigvee_{\{\sigma \mid \sigma \in \mathsf{Dom}(\mathsf{Obs}) \wedge \sigma_{\restriction \Sigma_i} = \sigma'\}} \mathsf{Obs}(\sigma)$*, for all* $\sigma' \in \mathsf{Dom}(\mathsf{Obs}_{\Sigma_i})$.

This definition is taken to mimic the behavior of parallel composition, *i.e.*, a component T_i accepts σ' if and only if there is some σ such that $\sigma_{\restriction \Sigma_i} = \sigma'$ and $\|_{i=1}^{n} T_i$ accepts σ.

▶ **Example 3.9.** Consider again the LTSs from Figure 1 and suppose we are given the following observation function for $T_1 \,\|\, T_2$: $\mathsf{Obs} : a \mapsto +; aa \mapsto -; abd \mapsto +$. The local observation functions we obtain for T_1 and T_2 are, respectively:

$$\mathsf{Obs}_{\{a,b,c\}} : a \mapsto +; aa \mapsto -; ab \mapsto + \qquad \mathsf{Obs}_{\{b,d\}} : \epsilon \mapsto +; bd \mapsto +.$$

The observation $\mathsf{Obs}(abd) = +$ requires both components to cooperate, hence $\mathsf{Obs}_{\{a,b,c\}}(ab) = +$ and $\mathsf{Obs}_{\{b,d\}}(bd) = +$. We derive $\mathsf{Obs}_{\{b,d\}}(\epsilon) = +$ from $\mathsf{Obs}(a) \vee \mathsf{Obs}(aa) = +$, since the projection of both these traces to $\{b, d\}$ is ϵ.

4 Distributions

In this section, we first discuss how we decompose the global alphabet into a *distribution*, i.e., a set of potentially overlapping local alphabets. We then give some properties of distributions and their relation to observation functions. Based on these, we explain how to extend a distribution to model a given observation function.

4.1 Distributions and Observations

In a model expressed as a parallel composition $\|_i T_i$, permuting symbols belonging to different local alphabets does not affect membership of the language. For example, in Figure 1, because *abcd* is in the language, we directly know that *abdc* is too, as both c and d are local to different components. To formalize this, we first formally define distributions as follows.

▶ **Definition 4.1** (Distribution). *A* distribution *of an alphabet* Σ *is a set* $\Omega = \{\Sigma_1, \ldots, \Sigma_n\}$ *such that* $\bigcup_{i=1}^{n} \Sigma_i = \Sigma$.

For the rest of this section, we fix an alphabet Σ, a distribution $\Omega = \{\Sigma_1, \ldots, \Sigma_n\}$ of Σ and an observation function Obs over Σ unless otherwise specified.

For a given distribution Ω, we define below the class of languages, called *product languages* over Ω, that can be represented over that distribution.

▶ **Definition 4.2** (Product language). $\mathcal{L}$ *is a* product language *over* Ω*, notation* $\Omega \models \mathcal{L}$*, iff there exists a family of languages* $\{\mathcal{L}_i\}_{1 \leq i \leq n}$*, where* $\mathcal{L}_i \subseteq \Sigma_i^\star$ *for all* $1 \leq i \leq n$*, such that* $\mathcal{L} = \|_{i=1}^{n}(\mathcal{L}_i, \Sigma_i)$.

▶ **Example 4.3.** In Example 3.5, it is clear by construction that $\{\{a, b, c\}, \{b, d\}\} \models \mathcal{L}(T_1 \,\|\, T_2)$. However, $\mathcal{L}(T_1 \,\|\, T_2)$ is not a product language over $\Omega_{singles} = \{\{a\}, \{b\}, \{c\}, \{d\}\}$ because any product language over $\Omega_{singles}$ should allow for permuting a and b and thus, would fail to capture the fact that b can only come after one a.

We recall the following key lemma for product languages.

▶ **Lemma 4.4** ([30], Lemma 5.2). *A language* $\mathcal{L}$ *is a product language over* Ω *if and only if* $\mathcal{L} = \|_{i=1}^{n}(\mathcal{L}_{\restriction \Sigma_i}, \Sigma_i)$.

We can now define product observations over Ω, *i.e.*, an observation that can be generated by a product language over Ω.

▶ **Definition 4.5** (Product observation). Obs *is a* product observation *over* Ω*, notation* $\Omega \models \mathsf{Obs}$*, iff there exists a language* $\mathcal{L}$ *such that* $\Omega \models \mathcal{L}$ *and* $\mathcal{L} \models \mathsf{Obs}$*. We conversely say that* Ω models Obs.

While Definition 4.5 does not prescribe how to find such a distribution given an observation function, it can be used to detect precisely when a current distribution is not consistent with observations and must be updated. This results in the following proposition linking local and global observations for a given distribution: an observation is a product observation over a distribution if and only if its projections according to the distribution hold the same information as the observation itself. The proof follows largely from Lemma 4.4.

▶ **Proposition 4.6.** $\Omega \models \mathsf{Obs}$ *if and only if for all traces* $\sigma \in \mathsf{Dom}(\mathsf{Obs})$ *it holds that* $\mathsf{Obs}(\sigma) = \bigwedge_{\Sigma_i \in \Omega} \mathsf{Obs}_{\Sigma_i}(\sigma_{\restriction \Sigma_i})$.

▶ **Example 4.7.** Following from Example 4.3, consider the following observation function based on $\mathcal{L}(T_1 \,\|\, T_2)$ from Example 3.5:

$$\mathsf{Obs} : \epsilon \mapsto +;\ a \mapsto +;\ ab \mapsto +;\ b \mapsto -; c \mapsto +;\ d \mapsto -.$$

Using the above proposition, we can verify that $\Omega_{singles} = \{\{a\}, \{b\}, \{c\}, \{d\}\} \not\models \mathsf{Obs}$. This is because $\mathsf{Obs}(b) = -$, whereas for all $\Sigma_i \in \Omega_{singles}$, $\mathsf{Obs}_{\Sigma_i}(b_{\restriction \Sigma_i}) = +$ since $ab_{\restriction \{b\}} = b$ causes $\mathsf{Obs}_{\{b\}}(b) = +$ and for other alphabets $b_{\restriction \Sigma_i} = \epsilon$ and $\mathsf{Obs}_{\Sigma_i}(\epsilon) = +$. In contrast, $\Omega_{\{a,b\}} = \{\{a, b\}, \{c\}, \{d\}\} \models \mathsf{Obs}$ since the alphabet $\{a, b\}$ allows for distinguishing observations b and ab.

In our algorithm, this check on local and global observation functions is used to trigger an update of the current distribution exactly when necessary.

Based on the above proposition, we now define *counter-examples to a distribution*. By definition of local observations, if $\mathsf{Obs}(\sigma) = +$, then $\mathsf{Obs}_{\Sigma_i}(\sigma_{\restriction \Sigma_i}) = +$. Hence, to obtain $\mathsf{Obs}(\sigma) \neq \bigwedge_{\Sigma_i \in \Omega} \mathsf{Obs}_{\Sigma_i}(\sigma_{\restriction \Sigma_i})$, we must have a globally negative observation σ_N and a set of globally positive observations whose projections to the local components match the projections of σ_N, indicating a mismatch between global and local observations.

▶ **Definition 4.8** (Counter-example to a distribution)**.** *A counter-example to* $\Omega \models \mathsf{Obs}$ *is a pair* $(\sigma_N, P) \in \mathsf{Dom}(\mathsf{Obs}) \times \mathsf{Dom}(\mathsf{Obs})^{\Omega}$ *with*

- σ_N *a negative observation* $\mathsf{Obs}(\sigma_N) = -$*;*
- P *a function that maps each* $\Sigma_i \in \Omega$ *to a positive observation* σ_{Σ_i}*, i.e.,* $\mathsf{Obs}(\sigma_{\Sigma_i}) = +$*, such that* $\sigma_{N \restriction \Sigma_i} = \sigma_{\Sigma_i \restriction \Sigma_i}$*.*

We call $\mathsf{img}(P)$ *the* positive image *of the counter-example. We write* $CED(\Omega, \mathsf{Obs})$ *for the set of such counter-examples.*

Although these counter-examples are not necessarily related to learning, we use the same terminology as in active learning. This is because the two concepts are directly linked in our case, as will be explained later.

▶ **Example 4.9.** Reusing the observation function Obs and the singleton distribution $\Omega_{singles}$ defined in Example 4.7, for every element of $CED(\mathsf{Obs}, \Omega_{singles})$ we have $\sigma_N = b.$ and $P(\{b\}) = ab.$ For the remaining elements of Ω, there are more choices: $\{a\}$ can be mapped to either ϵ or c; $\{c\}$ to either ϵ, a or ab; and $\{d\}$ to either ϵ, a, ab or c.

Proposition 4.6, specialized to our definition of counter-examples, yields the following corollary, which will be used in the following to detect that a distribution is a model.

▶ **Corollary 4.10.** $\Omega \models \mathsf{Obs} \iff CED(\Omega, \mathsf{Obs}) = \emptyset.$

4.2 Resolving a Counter-example

Given a distribution Ω and a fixed observation function Obs, one key question is how to extend Ω to a new distribution Ω' modelling Obs. This is a difficult problem, as new counter-examples can arise when extending a distribution. In this subsection, we explain how to resolve a single counter-example as a first step.

When a counter-example (σ_N, P) to $\Omega \models \mathsf{Obs}$ exists, it reveals a limitation in the distribution Ω: the projections of σ_N coincide with projections of elements in P, making them indistinguishable under the current components. To resolve such counter-examples, it is thus necessary and sufficient to augment Ω with new components that disrupt this matching. In the following, we will fix $(\sigma_N, P) \in CED(\Omega, \mathsf{Obs})$ as a counter-example to $\Omega \models \mathsf{Obs}$.

More precisely, for each pair of traces (σ_N, σ) with $\sigma \in P$, it suffices to identify a *discrepancy* between them. There are two types of discrepancies: *multiplicity discrepancies* and *order discrepancies*. A multiplicity discrepancy is a symbol occurring a different number of times in each trace. For this, given a trace σ, let $\Sigma^m(\sigma)$ denote the multiset of symbols occurring in σ. Note that $\Sigma^m(\sigma) = \Sigma^m(\sigma')$ if and only if σ is a permutation of σ'. The symmetric difference of multisets A and B is denoted $A \,\Delta\, B$.

▶ **Definition 4.11** (Multiplicity discrepancy). *Given a $\Sigma_i \in \Omega$, the set of* multiplicity discrepancies *for Σ_i is $\mathcal{D}_m^{\Sigma_i}(\sigma_N, P) = \Sigma^m(\sigma_N) \,\Delta\, \Sigma^m(P(\Sigma_i))$.*

We now define an *order discrepancy*, *i.e.*, a pair of symbols whose relative positions differ between the traces. We do this by considering whether symbols that are *not* a multiplicity discrepancy, *i.e.*, those appearing the same number of times in both traces, are permuted. We choose the permutation such that the relative order of identical symbols is maintained.

▶ **Definition 4.12** (Order discrepancy). *Given $\Sigma_i \in \Omega$, let $\theta = \Sigma \setminus (\Sigma^m(\sigma_N) \,\Delta\, \Sigma^m(P(\Sigma_i)))$ be the symbols on which σ_N and $P(\Sigma_i)$ agree and define $\sigma'_N = \sigma_{N\restriction\theta}$. Let π be the unique permutation such that $\sigma'_N = \pi(P(\Sigma_i)_{\restriction\theta})$ and $\sigma'_N[j] = \sigma'_N[k] \implies \pi(j) < \pi(k)$, for all $j < k$. The set of* order discrepancies *for Σ_i is then:*

$$\mathcal{D}_o^{\Sigma_i}(\sigma_N, P) = \{\{\sigma'_N[j], \sigma'_N[k]\} \mid k < j \wedge \pi(k) > \pi(j)\}$$

Multiplicity and order discrepancies can be found in linear and quadratic time, respectively. Finally, we define the *discrepancies* for a counter-example as sets that contain at least a discrepancy of either type for each alphabet in Ω.

▶ **Definition 4.13** (Discrepancy set). *A set $\delta \subseteq \Sigma$ is a* discrepancy *for (σ_N, P) iff for all $\Sigma_i \in \Omega$, either $\mathcal{D}_m^{\Sigma_i}(\sigma_N, P) \cap \delta \neq \emptyset$ or there is $\delta^{\Sigma_i} \in \mathcal{D}_o^{\Sigma_i}(\sigma_N, P)$ such that $\delta^{\Sigma_i} \subseteq \delta$. We write $\mathcal{D}(\sigma_N, P)$ for the set of all discrepancies for the counter-example (σ_N, P).*

For a set of counter-examples $\{ce_1, \ldots, ce_n\}$, we write $\mathcal{D}(\{ce_1, \ldots, ce_n\}) = \{\{\delta_1, \ldots, \delta_n\} \mid \forall i.\, \delta_i \in \mathcal{D}(ce_i)\}$, representing all possible selections of one discrepancy per counter-example.

▶ **Example 4.14** (Multiplicity discrepancy). Following Example 4.9 with the singleton distribution $\Omega_{singles} = \{\{a\}, \{b\}, \{c\}, \{d\}\}$, consider the counter-example $(\sigma_N, P) = (b, (\{a\} \mapsto \epsilon, \{b\} \mapsto ab, \{c\} \mapsto \epsilon, \{d\} \mapsto \epsilon))$. We find the following multiplicity discrepancies: $\mathcal{D}_m^{\Sigma_i}(\sigma_N, P) = \{b\}$, for $\Sigma_i \in \{\{a\}, \{c\}, \{d\}\}$, because b occurs once in σ_N vs. zero times in $P(\Sigma_i)$, and $\mathcal{D}_m^{\{b\}}(\sigma_N, P) = \{a\}$. Hence, $\mathcal{D}(\sigma_N, P)$ includes all subsets of $\{a, b, c, d\}$ containing $\{a, b\}$. For a different counter-example such as $(\sigma_N, P') = (b, (\{a\} \mapsto c, \{b\} \mapsto ab, \{c\} \mapsto ab, \{d\} \mapsto c))$, we obtain $\mathcal{D}_m^{\{a\}}(\sigma_N, P') = \mathcal{D}_m^{\{d\}}(\sigma_N, P') = \{b, c\}$ and $\mathcal{D}_m^{\{b\}}(\sigma_N, P') = \mathcal{D}_m^{\{c\}}(\sigma_N, P') = \{a\}$, so $\mathcal{D}(\sigma_N, P')$ includes any subset of $\{a, b, c, d\}$ that contains either $\{a, b\}$ or $\{a, c\}$.

▶ **Example 4.15** (Order discrepancy). Consider a singleton distribution $\Omega_{singles} = \{\{a\}, \{b\}, \{c\}\}$ with $\mathsf{Obs}(abc) = +$ and $\mathsf{Obs}(bac) = -$. This yields the counter-example $(\sigma_N, P) = (bac, (\{a\} \mapsto abc, \{b\} \mapsto abc, \{c\} \mapsto abc))$. For each $\Sigma_i \in \Omega_{singles}$, we find $\mathcal{D}_o^{\Sigma_i}(\sigma_N, P) = \{\{a, b\}\}$. Intuitively, these discrepancies reveals that singleton components allow for all permutations of a and b, but the observation function forbids some of them. Therefore, $\mathcal{D}(\sigma_N, P)$ includes all subsets of $\{a, b, c\}$ containing $\{a, b\}$.

We can now state that discrepancies are both sufficient and necessary additions to a distribution in order to eliminate their counter-examples.

▶ **Proposition 4.16.** *Suppose there exists $(\sigma_N, P) \in CED(\Omega, \mathsf{Obs})$. For each discrepancy $\delta \in \mathcal{D}(\sigma_N, P)$, $(\sigma_N, P) \notin CED(\Omega \cup \{\delta\}, \mathsf{Obs})$. Conversely, for any distribution Ω' of Σ where $\delta \not\subseteq \Sigma_i$, for all $\delta \in \mathcal{D}(\sigma_N, P)$ and all $\Sigma_i \in \Omega'$, (σ_N, P) is a counter-example to $\Omega' \models \mathsf{Obs}$.*

Figure 3 The different relations between the elements of two sets of counter-examples $CE \preccurlyeq CE'$.

4.3 Extending a Distribution to Model an Observation Function

Using the previous subsection as a basis, we leverage structural properties of distributions to restrict the possible counter-examples that can appear when updating the distribution. Finally, we devise an iterative process that is guaranteed to converge to a distribution modelling Obs.

Pre-ordering of Distributions

Distributions can be preordered by their "connecting power", i.e., by the extent to which they connect symbols together as part of the same alphabets.

▶ **Definition 4.17** (Connectivity preorder). *Given two distributions Ω and Ω' of alphabet Σ, we say that Ω is* less connecting *than Ω' and write $\Omega \preccurlyeq \Omega'$ when $\forall \Sigma_i \in \Omega.\, \exists \Sigma_j \in \Omega'.\, \Sigma_i \subseteq \Sigma_j$ (equivalently, Ω' is said to be* more connecting *than Ω). The relation is strict, written $\Omega \prec \Omega'$, when $\Omega' \not\preccurlyeq \Omega$. The relation $\preccurlyeq$ forms a preorder with finite chains.*

We relate this notion to the sets of counter-examples for a fixed observation function to show that adding connections in a distribution makes the counter-example set progress along a preorder. For this, we first define a notion of inclusion for counter-examples.

▶ **Definition 4.18** (Counter-example inclusion). *Consider two distributions Ω and Ω' of Σ, (σ_N, P) a counter-example to $\Omega \models \mathsf{Obs}$ and (σ'_N, P') a counter-example to $\Omega' \models \mathsf{Obs}$. We write $(\sigma_N, P) \subseteq (\sigma'_N, P')$ whenever $\sigma_N = \sigma'_N$ and $\mathsf{img}(P) \subseteq \mathsf{img}(P')$. The strict inclusion $(\sigma_N, P) \subset (\sigma'_N, P')$ holds whenever $\sigma_N = \sigma'_N$ and $\mathsf{img}(P) \subset \mathsf{img}(P')$.*

In its simplest form, progress means eliminating counter-examples from the current set of counter-examples $CED(\Omega, \mathsf{Obs})$. However, a counter-example ce might be replaced by new counter-examples ce' such that $ce \subseteq ce'$, which emerge when new connections are added to the distribution. Hence, progress means that some counter-examples are either eliminated or replaced by subsuming ones, as depicted in Figure 3.

▶ **Definition 4.19** (Counter-example set preordering). *Consider CE and CE' sets of counter-examples. We write $CE \preccurlyeq CE'$ when $\forall ce' \in CE'.\, \exists ce \in CE.\, ce \subseteq ce'$. We write $CE \prec CE'$ when, furthermore, either $ce \subset ce'$ or $CE \preccurlyeq CE' \setminus \{ce'\}$, for $ce \in CE$ and $ce' \in CE'$.*

▶ **Example 4.20.** We give a short example of the preorder on set of counter-examples inspired by our running example, that will later appear in Example 5.5:

$$\{(b, (\{a,b\} \mapsto cb, \{c\} \mapsto \epsilon, \{d\} \mapsto \epsilon))\} \preccurlyeq \{(b, (\{a,b\} \mapsto cb, \{b,c\} \mapsto ab, \{d\} \mapsto \epsilon))\}\ .$$

Notice that these two singleton sets' only difference is $\{c\} \mapsto \epsilon$ replaced by $\{b,c\} \mapsto ab$ such that the image of the first is strictly included in the image of the second.

Using the above definitions, we can prove that increasing the connecting power of a distribution ensures that the set of counter-example progresses.

▶ **Proposition 4.21.** *Consider two distributions Ω and Ω' of Σ. We have $\Omega \preccurlyeq \Omega' \Rightarrow CED(\Omega, \mathsf{Obs}) \preccurlyeq CED(\Omega', \mathsf{Obs})$.*

As an immediate consequence, whenever Ω has no counter-examples, any distribution Ω' that is more connecting than Ω will also have none, i.e., both distributions will model the same observations.

▶ **Corollary 4.22.** *Let Ω be a distribution of Σ such that $\Omega \models \mathsf{Obs}$. For any distribution Ω' of Σ such that $\Omega \preccurlyeq \Omega'$, we have $\Omega' \models \mathsf{Obs}$.*

Fixing the distribution

Using Propositions 4.16 and 4.21, from an initial distribution $\Omega \not\models \mathsf{Obs}$ we can create a more connecting one that entails a strict progression in counter-examples.

▶ **Corollary 4.23.** *Suppose that $CED(\Omega, \mathsf{Obs}) \neq \emptyset$. For $(\sigma_N, P) \in CED(\Omega, \mathsf{Obs})$, we pick a discrepancy $\delta_{(\sigma_N, P)} \in \mathcal{D}(\sigma_N, P)$. For any non-empty subset S of $CED(\Omega, \mathsf{Obs})$, let $\Omega' = \Omega \cup \{\delta_{ce} \mid ce \in S\}$. Then $\Omega \prec \Omega'$ and $CED(\Omega, \mathsf{Obs}) \prec CED(\Omega', \mathsf{Obs})$.*

▶ Remark 4.24. Corollary 4.23 gives us the freedom to select any discrepancy δ_{ce}, for each counter-example *ce*. We can select discrepancies that result in a least connecting distribution, which yields a locally optimal greedy strategy for progress. The intuition behind this choice is that it leads to: (1) more components that are individually smaller and easier to learn; and (2) fewer synchronizing actions between components, thus reducing the complexity of coordination among learners (see Section 5.1 for details).

By iteratively applying this Corollary, we can eliminate counter-examples until reaching a distribution that models the observations. This leads to the following convergence result:

▶ **Theorem 4.25.** *Suppose $\Omega \not\models \mathsf{Obs}$. The above process converges to a distribution $\Omega' \models \mathsf{Obs}$ such that $\Omega \prec \Omega'$ after finitely many of steps. For choosing $S = CED(\Omega, \mathsf{Obs})$ at each step, the number of steps is bounded by $|\mathcal{P}(\Sigma)|$.*

Canonical distributions: inducing a partial order

Distributions have few constraints: they only need to span the entire alphabet, which leaves room for redundancies. We propose to remove redundancies without affecting the distribution's connecting power, by removing alphabets completely contained within another.

▶ **Definition 4.26** (Canonical distribution)**.** *Consider a distribution $\Omega = \{\Sigma_1, \ldots, \Sigma_n\}$ and $Sub = \{\Sigma_i \in \Omega \mid \exists \Sigma_j \in \Omega.\ \Sigma_i \subset \Sigma_j\}$. The associated canonical distribution is $[\![\Omega]\!]_{\preccurlyeq} = \Omega \setminus Sub$.*

As one would expect $[\![\cdot]\!]_{\preccurlyeq}$ collapses equivalence classes of the preorder $\preccurlyeq$ (i.e., $\Omega \preccurlyeq \Omega'$ and $\Omega' \preccurlyeq \Omega$) to create a strict partial order. Canonical distributions allow minimizing the number of alphabets in the distribution while retaining the same connecting power. This means that counter-examples can be easily translated between a distribution and its canonical form, and hence the following proposition.

▶ **Proposition 4.27.** $CED([\![\Omega]\!]_{\preccurlyeq}, \mathsf{Obs}) = \emptyset \Leftrightarrow CED(\Omega, \mathsf{Obs}) = \emptyset$

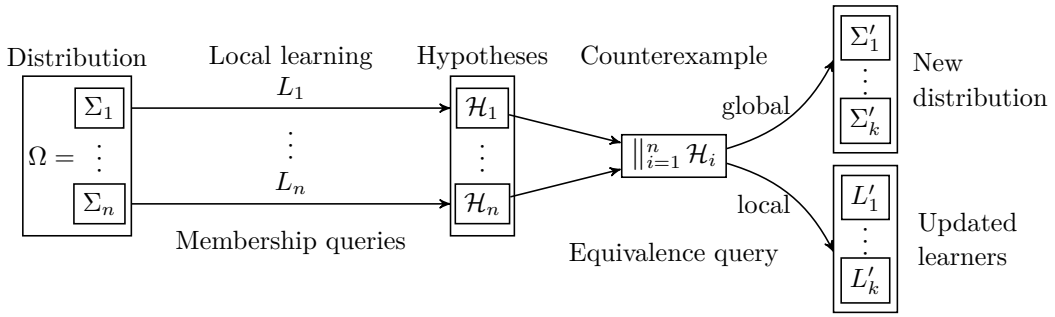


Figure 4 Overview of the algorithm.

5 Compositional Learning Algorithm

In this section, we present our algorithm to compositionally learn an unknown system $SUL = M_1 \parallel M_2 \parallel \ldots \parallel M_n$ consisting of the parallel composition of n LTSs, given only a Teacher for the whole SUL and knowledge of the global alphabet Σ_{SUL}.

A bird's eye view of the algorithm is provided in Figure 4. The key idea is to learn each component via a separate learner. Each learner L_i poses membership queries independently, which are suitably translated to queries for the global Teacher, until it produces a hypothesis $\mathcal{H}_i$. Hypotheses returned by local learners are combined to create a global equivalence query. Counter-examples obtained through equivalence queries $(\sigma, b) \in \Sigma^\star_{SUL} \times \{-, +\}$ are classified as either *global* or *local*. They are global when the updated observations $\mathsf{Obs}' = \mathsf{Obs} \cup \{(\sigma, b)\}$ and the current distribution Ω of Σ_{SUL} are such that $\Omega \not\models \mathsf{Obs}'$, and local otherwise. Global counter-examples are used to refine the distribution Ω, possibly creating components/learners. Local counter-examples are used to update the state of local learners.

We briefly recall the local learning procedure, which was introduced in previous work, before moving on to presenting the details of our main algorithm in Section 5.2.

5.1 Local learners

For each alphabet Σ_i in the distribution Ω, we spawn a learner L_i that is tasked with formulating a hypothesis for the corresponding component. A key feature of this learner is the ability to translate *local* membership queries into *global* ones for the entire SUL, and to interpret the results of global queries at the local level [31]. The main difficulty is that this translation is not always feasible: when a membership query contains synchronizing actions, cooperation with the other learners is required. This should be done in a way that is consistent with the current distribution, *i.e.*, if $\Omega \models \mathsf{Obs}$, then for any global membership query result $(\sigma, b) \in \Sigma^\star_{SUL} \times \{-, +\}$, we require $\Omega \models \mathsf{Obs} \cup \{(\sigma, b)\}$. Furthermore, due to the nature of local observation functions, it is possible that at first $\mathsf{Obs}_{\Sigma_i}(\sigma) = -$, and later this becomes $\mathsf{Obs}_{\Sigma_i}(\sigma) = +$ as the set of global observations is extended through counter-examples; the learners must account for this. In summary, local learners should:

- translate local queries to global ones, preserving $\Omega \models \mathsf{Obs}$. If a local query cannot be translated, the answer is "unknown";
- be able to handle "unknown" entries in the learning data structure (*e.g.*, an observation table), ensuring progress even in the presence of incomplete information; and
- be able to correct negative counter-examples on the basis of a later positive counter-example.

Our previous work [31] shows one way to implement such a learner as an extension of the $\mathtt{L}^\star$ algorithm [4], in which LTSs are represented as prefix-closed DFAs. However, we remark that the algorithms proposed in the present work are independent of the implementation of these local learners, as long as they satisfy the requirements above. Thus, our $\mathtt{L}^\star$-based implementation can be swapped out for other active learner algorithms, such as TTT [21].

5.2 Main algorithm

The main algorithm is presented in Algorithm 1. Initially, Obs is empty, and the distribution Ω contains singletons of the alphabet of the SUL. The algorithm iteratively performs the following steps.

Each learner is run in parallel until producing a hypothesis. Observations Obs are suitably updated to record the interactions with the Teacher. Next, the local hypotheses are composed in parallel to form $\mathcal{H}$, which is submitted to the Teacher as an equivalence query $\mathit{Teacher}(\mathcal{H})$. If the query returns no counter-example, the algorithm returns $\mathcal{H}$ and terminates. Otherwise, the returned counter-example (σ_{cex}, b) is added to Obs.

Crucially, when (σ_{cex}, b) is a global counter-example, it corresponds exactly to counter-examples to the distribution (Definition 4.8), as shown in the following lemma.

▶ **Lemma 5.1.** *Given a global counter-example (σ, b), let $\mathsf{Obs}' = \mathsf{Obs} \cup \{(\sigma, b)\}$:*

- *if $b = -$, then there is $P \in \mathsf{Dom}(\mathsf{Obs})^\Omega$ such that $(\sigma, P) \in \mathit{CED}(\Omega, \mathsf{Obs}')$.*
- *else, there is $\sigma_N \in \mathsf{Dom}(\mathsf{Obs})$, $S \subseteq \Omega$ and $P \in \mathsf{Dom}(\mathsf{Obs})^{\Omega \setminus S}$ such that $(\sigma_N, P \cup (\Sigma_i \mapsto \sigma)_{\Sigma_i \in S}) \in \mathit{CED}(\Omega, \mathsf{Obs}')$*

Furthermore, all of the elements of $\mathit{CED}(\Omega, \mathsf{Obs}')$ have the above structure.

Therefore, based on this lemma, the distribution is augmented with discrepancies for a chosen subset S of distribution counter-examples, following Corollary 4.23. This process eventually converges to provide Ω such that $\Omega \models \mathsf{Obs}$ (Theorem 4.25). The new distribution is then optimized by making it canonical (Definition 4.26) and, if desired, increasing its connectivity. The optimization step does not affect counter-example-freeness (by Proposition 4.27 and Corollary 4.22) and may be used to reduce synchronizations, which improves performance (see Section 6). New learners are then started over the updated alphabets.[1]

If, instead, the counter-example is local, its projections are forwarded to the local learners, and the next iteration starts.

▶ Remark 5.2. We leave the selection of counter-example set S as an implementation choice. While $S = \mathit{CED}(\Omega, \mathsf{Obs})$ maximizes counter-example elimination, finding all counter-examples may be expensive. In our implementation, we process just one counter-example at a time, which in practice often yields a valid distribution after a single update step.

Our main theorem for this section states that the algorithm terminates and returns a correct model of the SUL.

▶ **Theorem 5.3.** *Let $\mathit{SUL} = M_1 \parallel M_2 \parallel \ldots \parallel M_n$ consist of n parallel LTSs. Algorithm 1 terminates and returns $\mathcal{H}_1, \mathcal{H}_2, \ldots, \mathcal{H}_k$ such that $\mathcal{H}_1 \parallel \mathcal{H}_2 \parallel \ldots \parallel \mathcal{H}_k \equiv \mathit{SUL}$.*

▶ Remark 5.4. We make no claims regarding the number of components returned by the algorithm, nor whether $\mathcal{H}_i \equiv M_i$, for all i. This is because the final distribution may vary depending on counter-example and discrepancy choices. Moreover, different sets of

[1] In practice, learners leverage Obs to partially initialize their observation tables.

Algorithm 1 Main learning algorithm.

Input: The alphabet of the SUL Σ_{SUL} and the teacher *Teacher*.
Init: $\mathsf{Obs} = \emptyset$, $\Omega = \{\{a\} \mid a \in \Sigma_{SUL}\}$, $\mathbb{L} = \{\text{new learner } L_i \text{ on } \Sigma_i \mid \Sigma_i \in \Omega\}$
while *True* **do**
  foreach $L_i \in \mathbb{L}$ *in parallel* **do**
    Locally learn using *Teacher* until an hypothesis $\mathcal{H}_i$ is returned
    $\mathsf{Obs} \leftarrow \mathsf{Obs} \cup \{(\sigma, b) \mid (\sigma, b) \in \text{global membership queries of } L_i\}$
  $\mathcal{H} = \|_{i=1}^{|\Omega|} \mathcal{H}_i$
  $CEX = Teacher(\mathcal{H})$
  if *CEX is empty* **then**
    Return $\mathcal{H}_1, \ldots, \mathcal{H}_{|\Omega|}$
  else if $CEX = (\sigma_{cex}, b)$ **then**
    $\mathsf{Obs} \leftarrow \mathsf{Obs} \cup \{(\sigma_{cex}, b)\}$
    if (σ_{cex}, b) *is global* **then**
      while $CED(\Omega, \mathsf{Obs}) \neq \emptyset$ **do**
        Pick $S \subseteq CED(\Omega, \mathsf{Obs})$ non-empty
        $\Omega \leftarrow \Omega \cup \Omega'$ for some $\Omega' \in \mathcal{D}(S)$
      $Optimize(\Omega)$
      $\mathbb{L} \leftarrow \{\text{new learner } L_i \text{ on } \Sigma_i \mid \Sigma_i \in \Omega\}$
    else
      foreach $L_i \in \mathbb{L}$ *learning on* Σ_i **do**
        Forward $\sigma_{cex\restriction\Sigma_i}$ to L_i

component LTSs can result in the same parallel composition (see [31, Remark 2]). While the composition $\mathcal{H}_1 \,\|\, \mathcal{H}_2 \,\|\, \ldots \,\|\, \mathcal{H}_k$ may not be canonical (i.e., minimal), each $\mathcal{H}_i$ is guaranteed to be a canonical model of the local observation function Obs_{Σ_i}, as each local learner is a (slightly modified) instance of $\mathtt{L}^\star$, for which we have minimality guarantees.

▶ **Example 5.5** (Example run)**.** We give an example run where the target SUL is the model of Figure 1. For the sake of simplicity, we focus on the global counter-examples and the subsequent distribution updates, considering only one distribution counter-example per step. Moreover, we consistently select a smallest discrepancy for each counter-example as our greedy strategy to minimize the connectivity of the resulting distribution.

We start from $\Omega_{singles} = \{\{a\}, \{b\}, \{c\}, \{d\}\}$. The local alphabets initially contain only one symbol, so local learners will make membership queries about traces containing exclusively that symbol. This leads to the components depicted below.

The first global counter-example is $(ab, +)$, yielding several counter-examples to $\Omega_{singles} \models \mathsf{Obs}$, of which we consider $(b, (\{a\} \mapsto \epsilon, \{b\} \mapsto ab, \{c\} \mapsto \epsilon, \{d\} \mapsto \epsilon))$. The smallest discrepancy for this counter-example is $\{a, b\}$. We use it to update the distribution and obtain $\Omega_{ab} = \{\{a, b\}, \{c\}, \{d\}\}$,[2], which models the current observations. The new component over $\{a, b\}$ is then learned locally, producing (the $\{c\}$ and $\{d\}$ components are unchanged):

[2] We made the distribution canonical (Definition 4.26) and removed the $\{a\}$ and $\{b\}$ components.

The next global counter-example $(cb, +)$, leads to distribution counter-example $(b, (\{a, b\} \mapsto cb, \{c\} \mapsto \epsilon, \{d\} \mapsto \epsilon))$. Its smallest discrepancy is $\{b, c\}$ and the new distribution is $\Omega_{ab,bc} = \{\{a, b\}, \{b, c\}, \{d\}\}$. Although the counter-example has been handled, $\Omega_{ab,bc}$ does not model the observations, as $CED(\Omega_{ab,bc}, \mathsf{Obs})$ contains $(b, (\{a, b\} \mapsto cb, \{b, c\} \mapsto ab, \{d\} \mapsto \epsilon))$. Its smallest discrepancy $\{a, b, c\}$ gives $\Omega_{abc} = \{\{a, b, c\}, \{d\}\}$, modelling the observations.

To finish our example, the next global counter-example is $(abd, +)$. The corresponding distribution counter-example is $(d, (\{a, b, c\} \mapsto \epsilon, \{d\} \mapsto abd))$. There are two smallest discrepancies for this counter-example: $\{a, d\}$ and $\{b, d\}$. Selecting $\{b, d\}$ leads to $\{\{a, b, c\}, \{b, d\}\}$, which models the target language and exactly corresponds to the decomposition of Figure 1. Selecting $\{a, d\}$ creates unnecessary connections, resulting (after some omitted steps) in either $\{\{a, b, c\}, \{a, d\}, \{b, d\}\}$ or $\{\{a, b, c\}, \{a, b, d\}\}$ as a final distribution.

Our current implementation selects either discrepancy as both are locally optimal. Finding efficient ways to explore multiple discrepancy choices for globally optimal distributions remains an open challenge for future work.

6 Experiments

We evaluate the effectiveness of our approach in terms of savings in membership and equivalence queries. We did not expect to gain efficiency (absolute execution time) over the mature available tools, because our current implementation of local learners [31] interfaces with an external SAT solver for forming local hypotheses. To start with, we extended the tool Coal [31] into Coal$A$ (for COmpositional Automata Learner with Alphabet refinement), by adding the ability to refine the alphabets based on global counter-examples. The tool is based on LearnLib 0.18.0 [22]. As discussed in Section 5.2, the theory allows optimizing the distribution. Our implementation of this is based on greedily finding a (clique) edge cover in the hypergraph (Σ, Δ), where Δ is the set of all discrepancies found thus far. Since the learners L_i perform better when their alphabet contains more local actions, our algorithm tries to optimise for this. We sometimes also merge components to convert synchronizations into local actions.

To validate our approach, we experiment with learning LTSs obtained from three sources:

1. 300 randomly generated LTSs from our previous work [31], varying in structure and size.
2. 328 LTSs obtained from Petri nets that are provided by the Hippo tool [40] website;[3] these are often more sequential in nature than our other models.
3. Two scalable realistic models, namely *CloudOpsManagement* from the 2019 Model Checking Contest [3] and a *producers/consumers* model [41, Fig. 8].

Using a machine with four Intel Xeon 6136 processors and 3TB of RAM running Ubuntu 20.04, we apply each of three approaches: (i) our black-box compositional approach (Coal$A$), (ii) compositional learning with given alphabets (Coal) [31], and (iii) monolithic learning with $\mathtt{L}^\star$ (as implemented in LearnLib). Coal can be viewed as an idealized (best-case) baseline where the knowledge of the system decomposition is already available. Each run has a time-out of 30 minutes. We record the number of membership and equivalence queries posed to the Teacher, which we assume answers queries in constant time. This eliminates

[3] `https://hippo.uz.zgora.pl/`

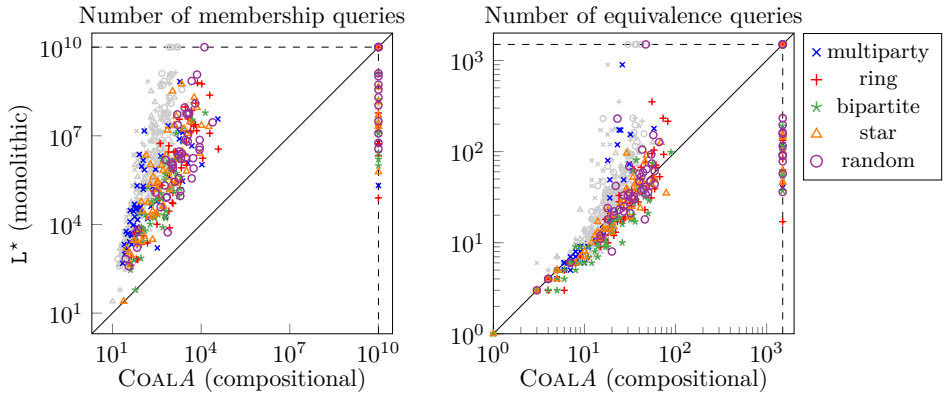


Figure 5 Performance of $L^\star$ and compositional learning on 300 randomly generated models. Dashed lines indicate time-outs. Results obtained with Coal are in gray.

variations in runtime caused by the Teacher, and ensures that local negative counter-examples (see Section 5.1) are always eventually corrected. Resolving this limitation is orthogonal to the current work and left for future research. A complete replication package is at [18].

Random models

Figure 5 shows the results on our random models. The colors indicate various communication structures (see [31] for details). As a reference, the results obtained with Coal are given in gray. We observe that Coal$A$ requires significantly fewer membership queries than $L^\star$ (note the logarithmic scale) and is closer to the theoretical optimum of Coal; the result show 5-6 orders of magnitude of improvement in a large number of concurrent systems. The number of equivalence queries required by Coal$A$ is typically slightly higher, but results of larger instances suggest that Coal$A$ scales better than its monolithic counterpart. The data shows that also in the case of equivalence queries, it is not uncommon to gain an order of magnitude of saving by using our approach. We note that Coal$A$ timeouts occur not due to a high query count, but because local learners use SAT solving to construct minimal hypotheses from observation tables that contain unknowns. This process can be computationally expensive.

Petri Net models

The results of learning the Hippo models are given in Figure 6. Here we are not able to run Coal, since the component alphabets are not known. Coal$A$ does not perform as well as on random models, in particular it requires more equivalence queries than monolithic $L^\star$. This is explained by the fact that these Petri nets contain mostly sequential behavior, *i.e.*, the language roughly has the shape $\mathcal{L}_1 \cdot a \cdot \mathcal{L}_2$ for some languages $\mathcal{L}_1 \subseteq \Sigma_1^*$ and $\mathcal{L}_2 \subseteq \Sigma_2^*$. Even though our learner is able to find the decomposition $\{\Sigma_1 \cup \{a\}, \Sigma_2 \cup \{a\}\}$, we do not gain much due to the absence of concurrent behavior. In the Hippo benchmark set, Coal$A$ typically finds between two and nine components.

Realistic models

Finally, Table 1 shows the results of learning two scalable models with relevant parameters indicated. No timeouts are reported in the table because all systems were successfully learned within the time limit. Coal$A$ scales well as the SUL size increases, requiring roughly a

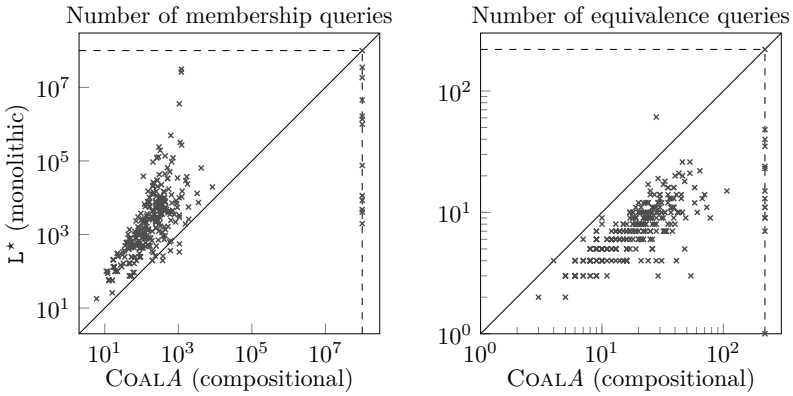


Figure 6 Performance of L* and compositional learning on Hippo models. Dashed lines indicate time-outs or out-of-memory.

constant factor more queries than Coal for both CloudOps and producers/consumers. We remark that practically all runtime of Coal$A$ and Coal is spent in the local learners: they require expensive SAT queries to construct a local hypothesis. Improving the implementation of local learners would decrease these times significantly; we stress that this is orthogonal to the goal of the current work. Furthermore, in any practical scenario, the processing of queries by the Teacher forms the bottleneck and L* would be much slower than Coal$A$.

Table 1 Performance of Coal$A$ and L* for realistic composite systems. Reported runtimes are in seconds. The number of refinement iterations is listed under "it." and the number of components found under "com".

		Coal$A$					Coal				L*		
model	states	time	memQ	eqQ	it.	com	time	memQ	eqQ	com	time	memQ	eqQ
CloudOps W=1,C=1,N=3	690	225.21	7 686	106	67	9	1.27	880	24	5	3.29	2 740 128	88
CloudOps W=1,C=1,N=4	1 932	235.97	9 521	115	74	10	1.93	923	26	6	29.62	22 252 120	216
CloudOps W=2,C=1,N=3	3 858	232.87	25 968	98	63	8	357.51	9 812	29	5	12.69	12 574 560	99
CloudOps W=2,C=1,N=4	10 824	708.95	33 616	111	72	10	555.82	9 532	30	6	143.28	91 178 900	227
ProdCons K=3,P=2,C=2	1 664	3.84	685	26	23	5	1.40	301	13	5	3.05	2 141 165	43
ProdCons K=5,P=1,C=1	170	1.03	451	22	14	3	0.88	118	11	3	0.40	160 126	30
ProdCons K=5,P=2,C=1	662	1.59	379	24	17	4	0.93	158	13	4	2.79	2 523 625	91
ProdCons K=5,P=2,C=2	2 240	5.53	697	28	25	5	2.35	282	14	5	6.40	6 984 705	93
ProdCons K=5,P=3,C=2	8 750	17.82	1 305	31	30	6	6.33	604	15	6	72.87	60 186 235	187
ProdCons K=5,P=3,C=3	30 344	73.69	2 454	34	37	7	32.59	1 269	17	7	321.54	222 567 729	193
ProdCons K=7,P=2,C=2	2 816	5.26	715	29	25	5	2.20	307	15	5	16.92	15 792 997	135

7 Conclusion

We presented a novel active learning algorithm that automatically discovers component decompositions using only global observations. Unlike previous approaches, our technique handles a general synchronization scheme common in automata theory and process calculi, without any prior knowledge of component structure. We developed a theory of alphabet distributions and their relation to observations, formally characterizing counter-examples that indicate inconsistencies between distributions and observations, and providing systematic methods to resolve them. The algorithm spawns a local learner for each component, dynamically refining the distribution of the alphabet based on counter-examples. Our

CoalA implementation dramatically reduces membership queries and achieves better query scalability than monolithic learning on highly concurrent systems. Future work will focus on:

- developing a theory of counter-example and discrepancy selection to characterize optimal distributions;
- extending our approach to apartness-based learning [37] and more expressive formalisms such as register automata [9] or timed automata [39];
- leveraging our compositional techniques to analyze AI-generated code [32]; and
- improving the efficiency of the local learners [24].

References

1 Fides Aarts, Joeri de Ruiter, and Erik Poll. Formal models of bank cards for free. In *ICST*, pages 461–468, 2013. doi:10.1109/ICSTW.2013.60.

2 Fides Aarts, Julien Schmaltz, and Frits W. Vaandrager. Inference and abstraction of the biometric passport. In *ISoLA*, volume 6415, pages 673–686, 2010. doi:10.1007/978-3-642-16558-0_54.

3 Elvio Amparore et al. Presentation of the 9th Edition of the Model Checking Contest. In *TACAS*, volume 11429, pages 50–68, 2019. doi:10.1007/978-3-030-17502-3_4.

4 Dana Angluin. Learning regular sets from queries and counterexamples. *Information and computation*, 75(2):87–106, 1987. doi:10.1016/0890-5401(87)90052-6.

5 Franco Barbanera, Mariangiola Dezani-Ciancaglini, Ivan Lanese, and Emilio Tuosto. Composition and decomposition of multiparty sessions. *Journal of Logical and Algebraic Methods in Programming*, 119:100620, 2021. doi:10.1016/J.JLAMP.2020.100620.

6 Benedikt Bollig, Joost-Pieter Katoen, Carsten Kern, and Martin Leucker. Learning Communicating Automata from MSCs. *IEEE Transactions on Software Engineering*, 36(3):390–408, 2010. doi:10.1109/TSE.2009.89.

7 Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors. *Model-Based Testing of Reactive Systems, Advanced Lectures [The volume is the outcome of a research seminar that was held in Schloss Dagstuhl in January 2004]*, volume 3472, 2005. doi:10.1007/B137241.

8 Ilaria Castellani, Madhavan Mukund, and P. S. Thiagarajan. Synthesizing distributed transition systems from global specification. In *FSTTCS*, volume 1738, pages 219–231, 1999. doi:10.1007/3-540-46691-6_17.

9 Simon Dierl, Paul Fiterau-Brostean, Falk Howar, Bengt Jonsson, Konstantinos Sagonas, and Fredrik Tåquist. Scalable tree-based register automata learning. In *TACAS*, volume 14571, pages 87–108, 2024. doi:10.1007/978-3-031-57249-4_5.

10 Andrzej Ehrenfeucht and Grzegorz Rozenberg. Partial (set) 2-structures. part II: state spaces of concurrent systems. *Acta Informatica*, 27(4):343–368, 1990. doi:10.1007/BF00264612.

11 Tiago Ferreira, Harrison Brewton, Loris D'Antoni, and Alexandra Silva. Prognosis: closed-box analysis of network protocol implementations. In *SIGCOMM*, pages 762–774, 2021. doi:10.1145/3452296.3472938.

12 Paul Fiterau-Brostean, Ramon Janssen, and Frits W. Vaandrager. Combining Model Learning and Model Checking to Analyze TCP Implementations. In *CAV*, volume 9780, pages 454–471, 2016. doi:10.1007/978-3-319-41540-6_25.

13 Paul Fiterau-Brostean, Toon Lenaerts, Erik Poll, Joeri de Ruiter, Frits W. Vaandrager, and Patrick Verleg. Model learning and model checking of SSH implementations. In *SPIN*, pages 142–151, 2017. doi:10.1145/3092282.3092289.

14 Markus Frohme and Bernhard Steffen. Compositional learning of mutually recursive procedural systems. *International Journal on Software Tools for Technology Transfer*, 23(4):521–543, 2021. doi:10.1007/s10009-021-00634-y.

15 Dominik Fuchß, Haoyu Liu, Tobias Hey, Jan Keim, and Anne Koziolek. Enabling architecture traceability by llm-based architecture component name extraction. In *ICSA*, pages 1–12, 2025. doi:10.1109/ICSA65012.2025.00011.

16 E. Mark Gold. Complexity of automaton identification from given data. *Information and Control*, 37(3):302–320, 1978. doi:10.1016/S0019-9958(78)90562-4.

17 Roberto Guanciale and Emilio Tuosto. Realisability of pomsets via communicating automata. In *ICE*, volume 279, pages 37–51, 2018. doi:10.4204/EPTCS.279.6.

18 Léo Henry, Mohammad Mousavi, Thomas Neele, and Matteo Sammartino. Replication package for the paper "Compositional Active Learning of Synchronous Systems through Automated Alphabet Refinement", April 2025. doi:10.5281/zenodo.15170685.

19 C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall International, 2004.

20 Falk Howar and Bernhard Steffen. Active automata learning in practice - an annotated bibliography of the years 2011 to 2016. In *Machine Learning for Dynamic Software Analysis*, volume 11026, pages 123–148, 2018. doi:10.1007/978-3-319-96562-8_5.

21 Malte Isberner, Falk Howar, and Bernhard Steffen. The TTT algorithm: A redundancy-free approach to active automata learning. In *RV*, volume 8734, pages 307–322, 2014. doi:10.1007/978-3-319-11164-3_26.

22 Malte Isberner, Falk Howar, and Bernhard Steffen. The Open-Source LearnLib: A Framework for Active Automata Learning. In *CAV*, volume 9206, pages 487–495, 2015. doi:10.1007/978-3-319-21690-4_32.

23 Rainer Koschke. *Architecture Reconstruction*. Springer-Verlag, 2009. doi:10.1007/978-3-540-95888-8.

24 Loes Kruger, Sebastian Junges, and Jurriaan Rot. Small test suites for active automata learning. In *TACAS*, volume 14571, pages 109–129, 2024. doi:10.1007/978-3-031-57249-4_6.

25 Faezeh Labbaf, Jan Friso Groote, Hossein Hojjat, and Mohammad Reza Mousavi. Compositional learning for interleaving parallel automata. In *FoSSaCS*, volume 13992, pages 413–435, 2023. doi:10.1007/978-3-031-30829-1_20.

26 Bas Luttik. Unique parallel decomposition in branching and weak bisimulation semantics. *Theoretical Computer Science*, 612:29–44, 2016. doi:10.1016/j.tcs.2015.10.013.

27 Aryan Bastany Mahboubeh Samadi and Hossein Hojjat. Compositional learning for synchronous parallel automata. In *FASE*, volume 15693, pages 101–121, 2025. doi:10.1007/978-3-031-90900-9_6.

28 Stefan Marksteiner, Marjan Sirjani, and Mikael Sjödin. Automated passport control: Mining and checking models of machine readable travel documents. In *ARES*, 2024. doi:10.1145/3664476.3670454.

29 Joshua Moerman. Learning product automata. In *ICGI*, volume 93, pages 54–66, 2018. URL: http://proceedings.mlr.press/v93/moerman19a.html.

30 Madhavan Mukund. From global specifications to distributed implementations. In *Synthesis and Control of Discrete Event Systems*, pages 19–35, 2002. doi:10.1007/978-1-4757-6656-1_2.

31 Thomas Neele and Matteo Sammartino. Compositional automata learning of synchronous systems. In *FASE*, volume 13991, pages 47–66, 2023. doi:10.1007/978-3-031-30826-0_3.

32 Marc North, Amir Atapour-Abarghouei, and Nelly Bencomo. Code gradients: Towards automated traceability of llm-generated code. In *RE*, pages 321–329, 2024. doi:10.1109/RE59067.2024.00038.

33 Mathijs Schuts, Jozef Hooman, and Frits W. Vaandrager. Refactoring of legacy software using model learning and equivalence checking: An industrial experience report. In *IFM*, volume 9681, pages 311–325, 2016. doi:10.1007/978-3-319-33693-0_20.

34 Maurice H. ter Beek, Rolf Hennicker, and José Proença. Team automata: Overview and roadmap. In *COORDINATION*, volume 14676, pages 161–198, 2024. doi:10.1007/978-3-031-62697-5_10.

35 Maurice H. ter Beek, Rolf Hennicker, and José Proença. Overview and roadmap of team automata. *CoRR*, abs/2501.13589, 2025. doi:10.48550/arXiv.2501.13589.

36 Frits W. Vaandrager. Model learning. *Communications of the ACM*, 60(2):86–95, 2017. doi:10.1145/2967606.

37 Frits W. Vaandrager, Bharat Garhewal, Jurriaan Rot, and Thorsten Wißmann. A new approach for active automata learning based on apartness. In *TACAS*, volume 13243, pages 223–243, 2022. doi:10.1007/978-3-030-99524-9_12.

38 Pepe Vila, Pierre Ganty, Marco Guarnieri, and Boris Köpf. CacheQuery: Learning Replacement Policies from Hardware Caches. In *PLDI*, pages 519–532, 2020. doi:10.1145/3385412.3386008.

39 Masaki Waga. Active learning of deterministic timed automata with myhill-nerode style characterization. In *CAV*, pages 3–26, 2023. doi:10.1007/978-3-031-37706-8_1.

40 Remigiusz Wiśniewski, Grzegorz Bazydło, Marcin Wojnakowski, and Mateusz Popławski. Hippo-CPS: A Tool for Verification and Analysis of Petri Net-Based Cyber-Physical Systems. In *Petri Nets 2023*, volume 13929, pages 191–204, 2023. doi:10.1007/978-3-031-33620-1_10.

41 W.M. Zuberek. Petri net models of process synchronization mechanisms. In *SMC*, volume 1, pages 841–847, 1999. doi:10.1109/ICSMC.1999.814201.

A Proofs

A.1 Section 4

▶ **Proposition 4.6.** $\Omega \models \mathsf{Obs}$ *if and only if for all traces* $\sigma \in \mathsf{Dom}(\mathsf{Obs})$ *it holds that* $\mathsf{Obs}(\sigma) = \bigwedge_{\Sigma_i \in \Omega} \mathsf{Obs}_{\Sigma_i}(\sigma_{\restriction \Sigma_i})$.

Proof. We make the proof by double implication.

Suppose $\Omega \models \mathsf{Obs}$. By definition this means that there is $\Omega \models \mathcal{L}$ such that $\mathcal{L} \models \mathsf{Obs}$. Since L is a product language, Lemma 4.4 yields that $\mathcal{L} = \{\sigma \mid \forall 1 \leq i \leq n.\, \sigma_{\restriction \Sigma_i} \in \mathcal{L}_i\}$ where $\mathcal{L}_i = \{\sigma_{\restriction \Sigma_i} \mid \sigma \in \mathcal{L}\}$ for all i.

Now for any $\sigma \in \mathsf{Dom}(\mathsf{Obs})$, we wish to show that

$$\mathsf{Obs}(\sigma) = + \text{ iff } \bigwedge_{\Sigma_i \in \Omega} \mathsf{Obs}_{\Sigma_i}(\sigma_{\restriction \Sigma_i}) = +.$$

We separate the two implications.

If $\mathsf{Obs}(\sigma) = +$, then $\mathsf{Obs}_{\Sigma_i}(\sigma_{\restriction \Sigma_i}) = +$ for all i by definition of local observation functions and we have our result.

If $\bigwedge_{\Sigma_i \in \Omega} \mathsf{Obs}_{\Sigma_i}(\sigma_{\restriction \Sigma_i}) = +$, it means that for any i, $\mathsf{Obs}_{\Sigma_i}(\sigma_{\restriction \Sigma_i}) = +$. By definition of local observations, for all i there is $\sigma'_i \in \mathsf{Dom}(\mathsf{Obs})$ such that $\mathsf{Obs}(\sigma'_i) = +$ and $\sigma_{\restriction \Sigma_i} = \sigma'_{i \restriction \Sigma_i}$. As $\mathcal{L} \models \mathsf{Obs}$, $\sigma'_i \in \mathcal{L}$.

This implies that $\forall i.\, \exists \sigma'_i \in \mathcal{L} \wedge \sigma_{\restriction \Sigma_i} = \sigma'_{i \restriction \Sigma_i}$. Hence for all i, $\sigma_{\restriction \Sigma_i} \in \mathcal{L}_i$ by definition of $\mathcal{L}_i$. It follows by definition of $\mathcal{L}$ that $\sigma \in \mathcal{L}$. As $\mathcal{L} \models \mathsf{Obs}$, we have $\mathsf{Obs}(\sigma) = +$.

Conversely, suppose that $\forall \sigma \in \mathsf{Dom}(\mathsf{Obs}),\ \mathsf{Obs}(\sigma) = \bigwedge_{\Sigma_i \in \Omega} \mathsf{Obs}_{\Sigma_i}(\sigma_{\restriction \Sigma_i})$.

Consider the language $\mathcal{L} = \{\sigma \in \mathsf{Dom}(\mathsf{Obs}) \mid \mathsf{Obs}(\sigma) = +\}$ and, for all i, let $\mathcal{L}_i = \{\sigma \in \mathsf{Dom}(\mathsf{Obs}_{\Sigma_i}) \mid \mathsf{Obs}_{\Sigma_i}(\sigma) = +\}$. Clearly, we have that $\mathcal{L} \models \mathsf{Obs}$. Furthermore, by assumption, $\mathcal{L} = \{\sigma \mid \forall 1 \leq i \leq n, \sigma_{\restriction \Sigma_i} \in \mathcal{L}_i\}$. Hence $\Omega \models \mathcal{L}$ by definition. $\Omega \models \mathsf{Obs}$ follows. ◀

▶ **Corollary 4.10.** $\Omega \models \mathsf{Obs} \iff \mathit{CED}(\Omega, \mathsf{Obs}) = \emptyset$.

Proof. We make the proof by double implication.

When $\mathit{CED}(\Omega, \mathsf{Obs}) \neq \emptyset$ there is a counter-example (σ_N, P) to $\mathsf{Obs} \models \mathsf{Obs}$. In particular $\mathsf{Obs}(\sigma_N) = -$ and for all $\Sigma_i \in \Omega$ $\mathsf{Obs}_{\Sigma_i}(\sigma_{N \restriction \Sigma_i}) = \mathsf{Obs}_{\Sigma_i}(P(\Sigma_i)_{\restriction \Sigma_i}) = +$ by definition of a counter-example and local observation functions. Hence by Proposition 4.6, $\Omega \not\models \mathsf{Obs}$.

Conversely, when $CED(\Omega, \mathsf{Obs}) = \emptyset$ then for any $\sigma \in \mathsf{Dom}(\mathsf{Obs})$ such that $\mathsf{Obs}(\sigma) = -$ there is some i such that for any $\sigma' \in \mathsf{Dom}(\mathsf{Obs})$ verifying $\mathsf{Obs}_{\Sigma_i}(\sigma'_{\restriction \Sigma_i}) = \mathsf{Obs}_{\Sigma_i}(\sigma_{\restriction \Sigma_i})$, $\mathsf{Obs}(\sigma') = -$.

It follows that for any $\sigma \in \mathsf{Dom}(\mathsf{Obs})$ such that $\mathsf{Obs}(\sigma) = -$, $\bigwedge_{1 \leq i \leq n} \mathsf{Obs}_{\Sigma_i}(\sigma_{\restriction \Sigma_i} = -)$. As by definition of local observations, for any $\sigma \in \mathsf{Dom}(\mathsf{Obs})$ such that $\mathsf{Obs}(\sigma) = +$, $\bigwedge_{1 \leq i \leq n} \mathsf{Obs}_{\Sigma_i}(\sigma_{\restriction \Sigma_i} = +)$ we have $\Omega \models \mathsf{Obs}$ by Proposition 4.6. ◀

▶ **Proposition 4.16.** *Suppose there exists $(\sigma_N, P) \in CED(\Omega, \mathsf{Obs})$. For each discrepancy $\delta \in \mathcal{D}(\sigma_N, P)$, $(\sigma_N, P) \notin CED(\Omega \cup \{\delta\}, \mathsf{Obs})$. Conversely, for any distribution Ω' of Σ where $\delta \not\subseteq \Sigma_i$, for all $\delta \in \mathcal{D}(\sigma_N, P)$ and all $\Sigma_i \in \Omega'$, (σ_N, P) is a counter-example to $\Omega' \models \mathsf{Obs}$.*

Proof. Consider a discrepancy δ. We first prove that introducing δ into Ω is sufficient to remove the counter-example. As Obs is not changed by the change from Ω to Ω' we have to prove that there is no $\sigma_{\Sigma_i} \in \mathsf{img}(P)$ such that $\sigma_{N \restriction \delta} = \sigma_{\Sigma_i \restriction \delta}$.

Fix $\sigma_{\Sigma_i} \in \mathsf{img}(P)$.

- If $\mathcal{D}_m^{\Sigma_i}(\sigma_N, P) \cap \delta \neq \emptyset$ then there is $a \in \delta$ that occurs in different multiplicities in σ_N and σ_{Σ_i}. Hence, a also occurs in different multiplicities in $\sigma_{N \restriction \delta}$ and $\sigma_{\restriction \delta}$. Hence, $\sigma_{\restriction \delta} \neq \sigma_{i \restriction \delta}$.
- Otherwise, following the notations of Definition 4.12, $\sigma'_N = \pi(\sigma_{\Sigma_i \restriction \theta})$ and for all $j < k$ such that $\sigma'_N[j] = \sigma'_N[k]$ it holds that $\pi(j) < \pi(k)$. By preserving the order of equal symbols, we maintain that $\sigma[j] = a$ is the l-th occurrence of a in σ'_N iff $\sigma'[\pi(j)] = a$ is the l-th occurrence of a in σ_i.
  Now $\{a, b\} = \{\sigma'_N[j], \sigma'_N[k]\} \subseteq \delta$ by definition of a discrepancy and $\{a, b\}$ is created from a so-called *inversion* in π: a pair (j, k) such that $j < k$ and $\pi(j) > \pi(k)$. By our assumption on π, we know that $a \neq b$ and furthermore that in σ' at least one more copy of b precedes this a (namely the b at $\sigma'[\pi(k)]$). As a result of this and the fact that $\{a, b\} \subseteq \delta$, we have $\sigma'_{N \restriction \delta} \neq \sigma_{\Sigma_i \restriction \theta \restriction \delta}$. It follows directly that $\sigma_{N \restriction \delta} \neq \sigma_{\Sigma_i \restriction \delta}$.

We now prove that it is necessary. Consider Ω' defined as above and fix $\Sigma_j \in \Omega'$. In the following, we show that there is $\sigma \in \mathsf{img}(P)$ such that $\sigma_{N \restriction \Sigma_j} = \sigma_{\restriction \Sigma_j}$.

For this, as there is no discrepancy $\delta \in \mathcal{D}(\sigma_N, P)$ subset to Σ_j we know that there is at least one $\Sigma_i \in \Omega$ such that no order nor multiplicity discrepancy for σ δ_{Σ_i} is a subset of Σ_j. Because of this, $\Sigma^m(\sigma_N) \ \Delta \ \Sigma^m(P(\Sigma_i)) \cap \Sigma_j = \emptyset$. Hence $P(\Sigma_i)_{\restriction \Sigma_j}$ and $\sigma_{N \restriction \Sigma_j}$ are equal up to permutation. Consider any permutation π' such that $\sigma'_N = \pi'(\sigma_i)$ (following again the notations of Definition 4.12). Without loss of generality, we do not permute the position of equal symbols, which fixes a unique permutation $\pi' = \pi$. For any $\{\{\sigma'_N[j], \sigma'_N[k]\} \mid j < k \wedge \pi(j) > \pi(k)\}$, we know (as these are elements of $\mathcal{D}_o^{\Sigma_i}(\sigma_N, P)$) that we don't have $\{\sigma'_N[j], \sigma'_N[k]\} \subseteq \Sigma_j$. It follows that the restriction of π to the indices corresponding to symbols in Σ_j is non-decreasing and thus the identity. It follows that $\sigma_{N \restriction \Sigma_j} = P(\Sigma_i)_{\restriction \Sigma_j}$. ◀

▶ **Proposition 4.21.** *Consider two distributions Ω and Ω' of Σ. We have $\Omega \preccurlyeq \Omega' \Rightarrow CED(\Omega, \mathsf{Obs}) \preccurlyeq CED(\Omega', \mathsf{Obs})$.*

Proof. We first prove that $CED(\Omega, \mathsf{Obs}) \preccurlyeq CED(\Omega', \mathsf{Obs})$. Suppose $\Omega \preccurlyeq \Omega'$. If $CED(\Omega', \mathsf{Obs}) = \emptyset$ then we have our result. Else, take $(\sigma_N, P) \in CED(\Omega', \mathsf{Obs})$.

We know that for each $\Sigma_i \in \Omega$, there is $\Sigma_j^i \in \Omega'$ such that $\Sigma_i \subseteq \Sigma_j^i$. Fix such a Σ_j^i. We have that $\sigma_{N \restriction \Sigma_i} = P(\Sigma_j^i)_{\restriction \Sigma_i}$ as $\Sigma_i \subseteq \Sigma_j^i$ and $(\sigma_N, P) \in CED(\Omega', \mathsf{Obs})$.

By doing this for all $\Sigma_i \in \Omega$, we get $(\sigma_N, (P(\Sigma_j^i))_{\Sigma_i \in \Omega}) \in CED(\Omega, \mathsf{Obs})$ and by definition $(\sigma_N, (P(\Sigma_j^i))_{\Sigma_i \in \Omega}) \subseteq (\sigma_N, P)$. ◀

▶ **Corollary 4.23.** *Suppose that* $CED(\Omega, \mathsf{Obs}) \neq \emptyset$. *For* $(\sigma_N, P) \in CED(\Omega, \mathsf{Obs})$, *we pick a discrepancy* $\delta_{(\sigma_N,P)} \in \mathcal{D}(\sigma_N, P)$. *For any non-empty subset* S *of* $CED(\Omega, \mathsf{Obs})$, *let* $\Omega' = \Omega \cup \{\delta_{ce} \mid ce \in S\}$. *Then* $\Omega \prec \Omega'$ *and* $CED(\Omega, \mathsf{Obs}) \prec CED(\Omega', \mathsf{Obs})$.

Proof. $\Omega \preccurlyeq \Omega'$ follows directly from the definition of Ω': all elements of Ω are preserved. Furthermore, for $ce \in S$, for all $\Sigma_i \in \Omega$ we have $\delta_{ce} \not\subseteq \Sigma_i \in \Omega$ as $\sigma_{N\restriction\delta} \neq \sigma_{\Sigma_i\restriction\delta}$ and $\sigma_{N\restriction\Sigma_i} = \sigma_{\Sigma_i\restriction\Sigma_i}$. Hence $\Omega' \not\preccurlyeq \Omega$ and $\Omega \prec \Omega'$. From Proposition 4.21 we get that $CED(\Omega, \mathsf{Obs}) \preccurlyeq CED(\Omega', \mathsf{Obs})$. Furthermore, we know by Proposition 4.16 that for a fixed $ce \in S$, $CED(\Omega \cup D_{ce}, \mathsf{Obs})$ does not contain ce, ensuring that $CED(\Omega, \mathsf{Obs}) \prec CED(\Omega \cup \delta_{ce}, \mathsf{Obs})$. As furthermore $CED(\Omega \cup \delta_{ce}, \mathsf{Obs}) \preccurlyeq CED(\Omega', \mathsf{Obs})$ by construction, we have that $CED(\Omega, \mathsf{Obs}) \prec CED(\Omega', \mathsf{Obs})$. ◀

▶ **Theorem 4.25.** *Suppose* $\Omega \not\models \mathsf{Obs}$. *The above process converges to a distribution* $\Omega' \models \mathsf{Obs}$ *such that* $\Omega \prec \Omega'$ *after finitely many of steps. For choosing* $S = CED(\Omega, \mathsf{Obs})$ *at each step, the number of steps is bounded by* $|\mathcal{P}(\Sigma)|$.

Proof. In case of convergence, $\Omega \prec \Omega'$ is ensured directly by each step of the process by Corollary 4.23. We now prove that convergence occurs in a finite amount of steps. Each step eliminates all counter-examples that are part of the non-empty set S chosen, as proven in Proposition 4.16 and Corollary 4.23. As $CED(\Omega, \mathsf{Obs}) \prec CED(\Omega', \mathsf{Obs})$ the only possible new counter-examples are ones that contain strictly counter-examples of $CED(\Omega, \mathsf{Obs})$. It follows that iterating this process makes strict progress in the following sense: either counter-examples are eliminated or they are replaced by ones with a strictly larger positive image. Because the positive image size of counter-examples is bounded by the size of the distribution, itself bounded by $|\mathcal{P}(\Sigma)|$, we know that this converges in a finite number of steps.

If $S = CED(\Omega, \mathsf{Obs})$ at each step, we furthermore have that the size of the smallest positive image of a counter-example increases by at least one at each step, which bounds the number of steps by $|\mathcal{P}(\Sigma)|$. ◀

▶ **Proposition 4.27.** $CED([\![\Omega]\!]_{\preccurlyeq}, \mathsf{Obs}) = \emptyset \Leftrightarrow CED(\Omega, \mathsf{Obs}) = \emptyset$

Proof. We make the proof for a fixed $\Sigma_i \in \mathit{Sub}$, the result following by induction. By definition $\Omega' \preccurlyeq \Omega$, which entails that $CED(\Omega, \mathsf{Obs}) \preccurlyeq CED(\Omega', \mathsf{Obs})$. From there it follows that $CED(\Omega, \mathsf{Obs}) = \emptyset \Rightarrow CED(\Omega', \mathsf{Obs}) = \emptyset$. To get the other one, we reason by contrapositive. Suppose $CED(\Omega, \mathsf{Obs}) \neq \emptyset$ and consider (σ_N, P) in it. Then clearly, taking $P' = (P(\Sigma_j))_{j\neq i}$, $(\sigma_N, P') \in CED(\Omega', \mathsf{Obs})$ and we have our result. ◀

A.2 Section 5

▶ **Lemma 5.1.** *Given a global counter-example* (σ, b), *let* $\mathsf{Obs}' = \mathsf{Obs} \cup \{(\sigma, b)\}$:

- *if* $b = -$, *then there is* $P \in \mathsf{Dom}(\mathsf{Obs})^\Omega$ *such that* $(\sigma, P) \in CED(\Omega, \mathsf{Obs}')$.
- *else, there is* $\sigma_N \in \mathsf{Dom}(\mathsf{Obs})$, $S \subseteq \Omega$ *and* $P \in \mathsf{Dom}(\mathsf{Obs})^{\Omega\setminus S}$ *such that* $(\sigma_N, P \cup (\Sigma_i \mapsto \sigma)_{\Sigma_i \in S}) \in CED(\Omega, \mathsf{Obs}')$

Furthermore, all of the elements of $CED(\Omega, \mathsf{Obs}')$ *have the above structure.*

Proof. We know that $\Omega \models \mathsf{Obs}$ and $\Omega \not\models \mathsf{Obs}'$ since the counter-example is global. From Corollary 4.10 we get that $CED(\Omega, \mathsf{Obs}) = \emptyset$ and $CED(\Omega, \mathsf{Obs}') \neq \emptyset$.

Consider $(\sigma_N, P) \in CED(\Omega, \mathsf{Obs}')$. Since it is not a counter-example to $\Omega \models \mathsf{Obs}$, σ must appear in it. If $b = -$ we must have that $\sigma_N = \sigma$ by definition of counter-examples to a distribution. Similarly, if $b = +$ then $\sigma \in \mathsf{img}(P)$. Observing that $\mathsf{Dom}(\mathsf{Obs}') = \mathsf{Dom}(\mathsf{Obs}) \cup \{\sigma\}$, we obtain our result. ◀

▶ **Theorem 5.3.** *Let* $SUL = M_1 \parallel M_2 \parallel \ldots \parallel M_n$ *consist of* n *parallel LTSs. Algorithm 1 terminates and returns* $\mathcal{H}_1, \mathcal{H}_2, \ldots, \mathcal{H}_k$ *such that* $\mathcal{H}_1 \parallel \mathcal{H}_2 \parallel \ldots \parallel \mathcal{H}_k \equiv SUL$.

Proof. The correctness of the returned hypothesis is guaranteed by the Teacher.

The algorithm's termination is established by showing that the "while True" loop cannot execute indefinitely. This is because the two types of counter-examples can only occur finitely many times:

- For local counter-examples, when the distribution Ω is fixed, all the learners eventually converge to a hypothesis after encountering finitely many local counter-examples, as shown in [31, Theorem 2].
- For global counter-examples, each global counter-example leads to an updated distribution Ω' such that $\Omega \prec \Omega'$ (Theorem 4.25). This update can only happen finitely many times, as the top element of $\prec$ is $\{\Sigma_{SUL}\}$. ◀

Quantitative Language Automata

Thomas A. Henzinger
Institute of Science and Technology Austria (ISTA), Klosterneuburg, Austria

Pavol Kebis
Institute of Science and Technology Austria (ISTA), Klosterneuburg, Austria

Nicolas Mazzocchi
Slovak University of Technology in Bratislava, Slovakia

N. Ege Saraç
Institute of Science and Technology Austria (ISTA), Klosterneuburg, Austria

Abstract

A *quantitative word automaton* (QWA) defines a function from infinite words to values. For example, every infinite run of a limit-average QWA $\mathcal{A}$ obtains a mean payoff, and every word $w \in \Sigma^\omega$ is assigned the maximal mean payoff obtained by nondeterministic runs of $\mathcal{A}$ over w. We introduce *quantitative language automata* (QLAs) that define functions from language generators (i.e., implementations) to values, where a language generator can be nonprobabilistic, defining a set of infinite words, or probabilistic, defining a probability measure over infinite words. A QLA consists of a QWA and an aggregator function. For example, given a QWA $\mathcal{A}$, the infimum aggregator maps each language $L \subseteq \Sigma^\omega$ to the greatest lower bound assigned by $\mathcal{A}$ to any word in L. For boolean value sets, QWAs define boolean properties of traces, and QLAs define boolean properties of sets of traces, i.e., hyperproperties. For more general value sets, QLAs serve as a specification language for a generalization of hyperproperties, called *quantitative hyperproperties*. A nonprobabilistic (resp. probabilistic) quantitative hyperproperty assigns a value to each set (resp. distribution) G of traces, e.g., the minimal (resp. expected) average response time exhibited by the traces in G. We give several examples of quantitative hyperproperties and investigate three paradigmatic problems for QLAs: evaluation, nonemptiness, and universality. In the *evaluation* problem, given a QLA $\mathbb{A}$ and an implementation G, we ask for the value that $\mathbb{A}$ assigns to G. In the *nonemptiness* (resp. *universality*) problem, given a QLA $\mathbb{A}$ and a value k, we ask whether $\mathbb{A}$ assigns at least k to some (resp. every) language. We provide a comprehensive picture of decidability for these problems for QLAs with common aggregators as well as their restrictions to ω-regular languages and trace distributions generated by finite-state Markov chains.

2012 ACM Subject Classification Theory of computation → Automata over infinite objects; Theory of computation → Quantitative automata

Keywords and phrases Quantitative hyperproperties, quantitative automata, automata-based verification

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2025.21

Related Version *Full Version*: `https://arxiv.org/abs/2506.05158`

Funding This work was supported in part by the ERC-2020-AdG 101020093.

1 Introduction

The specification and verification of system properties traditionally take a boolean view. While this view is appropriate for correctness properties, it lacks the ability to reason about quantitative aspects of system behaviors, such as performance or robustness. Quantitative trace properties and quantitative word automata [12] address this gap: instead of partitioning the set of traces into correct and incorrect traces (as boolean properties do), they define

© Thomas A. Henzinger, Pavol Kebis, Nicolas Mazzocchi, and N. Ege Saraç;
licensed under Creative Commons License CC-BY 4.0

functions from system executions to richer value domains, e.g., the real numbers. Using such a formalism, we can specify the maximal or average response time of a server's execution, or how badly a system behavior violates a desired boolean property [19].

Many interesting system properties lie beyond the trace setting; especially security properties often refer to multiple traces. In the boolean case, they can be specified by hyperproperties [13], which are interpreted over sets of traces rather than over individual traces. For example, a prominent hyperproperty is observational determinism, which requires every pair of executions with matching observable inputs to have matching observable outputs. In general, while trace properties specify which system behaviors are correct, hyperproperties specify which system implementations are correct. In a similar vein, while quantitative trace properties and quantitative word automata describe system properties on the level of individual executions, *quantitative hyperproperties* are needed to express quantitative aspects of system-wide properties. For example, a quantitative hyperproperty may measure the maximal or average response time of an implementation instead of a single execution, or how badly an implementation violates a desired boolean property. In this paper, we introduce *quantitative language automata* (QLAs), an automaton model for the specification and verification of quantitative hyperproperties. In contrast to quantitative word automata, quantitative language automata can measure system-wide properties.

Quantitative word automata (QWAs) extend boolean ω-automata with weighted transitions. An infinite run yields an infinite sequence of weights, which are accumulated by a *run aggregator* (a.k.a. value function). Common run aggregators are Sup (the maximal weight along an infinite run), LimSup (the largest weight that occurs infinitely often), or LimInfAvg (the long-term average of an infinite sequence of weights). When a given infinite word yields more than one run, as is generally the case for nondeterministic automaton specifications, the possible run values are accumulated by a *word aggregator*. The most common word aggregator is Sup (the least upper bound of all values that can be achieved by resolving the nondeterministic choices), which generalizes the standard view that a single accepting run suffices to accept a word, but other word aggregators are possible. For example, the word aggregator LimSup assigns to each infinite word w the l.u.b. of the values realized by infinitely many runs over w. When the specification is probabilistic (rather than nondeterministic), the word aggregator $\mathbb{E}$ assigns to w an expected value for the automaton reading w.

Quantitative language automata extend quantitative word automata with a third kind of aggregator function, called *language aggregator*, which summarizes the values of all infinite words which are obtained from an implementation and defined by a so-called "language generator." A language generator can be nonprobabilistic, defining a set of infinite words, or probabilistic, defining a probability measure over infinite words. A QLA $\mathbb{A} = (h, \mathcal{A})$ consists of a language aggregator h and a QWA $\mathcal{A}$, and maps each language generator G to the value $h_{w \sim G} \mathcal{A}(w)$, where $\mathcal{A}(w)$ is the value assigned by the QWA $\mathcal{A}$ to the infinite word w, and $h_{w \sim G}$ denotes the accumulation over all words generated by G. When G is nonprobabilistic, we interpret the aggregator h over the words that belong to the language defined by G; when probabilistic, over the words generated with respect to the probability measure defined by G. Using the language aggregators Inf and Sup, language automata can measure the best-case and the worst-case values that can be obtained by all executions of an implementation. With language aggregators LimInf and LimSup, we can express the "almost" best- and worst-cases, considering only the values achieved by infinitely many words in the language. Finally, when G is probabilistic, the language aggregator $\mathbb{E}$ captures the average-case with respect to the its probability measure. Using all three aggregator functions, we can specify, for example, the expected value (taken over all possible implementation traces) of the best case (realizable by a nondeterministic specification) of the average weight along a run of the specification.

Table 1 Complexity of evaluation, nonemptiness, and universality for QLAs with language aggregator h, word aggregator g, and run aggregator f. **(a)** QLAs with $h, g \in \{\mathsf{Inf}, \mathsf{Sup}, \mathbb{E}\}$ (Sections 5 and 6). *Open-hard* means the problem is at least as hard as the universality problem for nondeterministic discounted-sum QWA [6, 12]. Universality of QLAs is dual to their nonemptiness, e.g., it is in PSpace when $h \in \{\mathsf{Inf}, \mathsf{Sup}, \mathbb{E}\}$ and $g = \mathsf{Sup}$ (see Remark 6.1). **(b)** QLAs with $h, g, f \in \{\mathsf{Inf}, \mathsf{Sup}, \mathsf{LimInf}, \mathsf{LimSup}\}$ with g or h in $\{\mathsf{LimInf}, \mathsf{LimSup}\}$ (Section 7).

(a)	Evaluation					Nonemptiness		
	$h = g$ $\in \{\mathsf{Inf}, \mathsf{Sup}\}$	$h \neq g$ $\in \{\mathsf{Inf}, \mathsf{Sup}\}$	$h = \mathbb{E}$ $g \in \{\mathsf{Inf}, \mathsf{Sup}\}$	$h \in \{\mathsf{Inf}, \mathsf{Sup}\}$ $g = \mathbb{E}$	$h = \mathbb{E}$ $g = \mathbb{E}$	$h \in \{\mathsf{Inf}, \mathsf{Sup}, \mathbb{E}\}$ $g = \mathsf{Sup}$	$g = \mathsf{Inf}$	$g = \mathbb{E}$
$f \in \{\mathsf{Inf}, \mathsf{Sup}, \mathsf{LimInf}, \mathsf{LimSup}\}$	PTime	PSpace	ExpTime	Undecidable	PTime	PTime	PSpace	Undecidable
$f \in \{\mathsf{LimInfAvg}, \mathsf{LimSupAvg}\}$	PTime	Undecidable	Undecidable	Undecidable	PTime	PTime	Undecidable	Undecidable
$f = \mathsf{DSum}$	PTime	Open-hard	Open-hard	Undecidable	PTime	PTime	Open-hard	Undecidable

(b)	Evaluation	Nonemptiness	Universality
$h, g, f \in \{\mathsf{Inf}, \mathsf{Sup}, \mathsf{LimInf}, \mathsf{LimSup}\}$ and at least one of h or g is in $\{\mathsf{LimInf}, \mathsf{LimSup}\}$	PSpace	PSpace	PSpace

The standard decision problems for automata extend naturally to our framework. Consider a language automaton $\mathbb{A}$ and a rational threshold k. Given a finite-state language generator G (i.e., an ω-regular automaton or a finite Markov chain), the evaluation problem asks to compute the value to which $\mathbb{A}$ maps G. The nonemptiness (resp. universality) problem asks whether $\mathbb{A}$ maps some (resp. every) language generator to a value of at least k, or strictly above k. We investigate these problems for QLAs over unrestricted language generators, as well as with regard to natural subclasses of language generators such as finite-state ones.

Contribution and Overview. Our main contribution is the definition and systematic study of QLAs within a three-dimensional framework of aggregator functions (run, word, and language aggregators), which supports the specification and verification of quantitative hyperproperties over nonprobabilistic and probabilistic language generators.

Section 2 presents our definitional framework and Section 3 gives several examples of QLAs for specifying quantitative hyperproperties. Section 4 introduces the QLA evaluation, nonemptiness, and universality problems that we study. Section 5 focuses on the evaluation problem of QLAs with word and language aggregators Inf, Sup, or $\mathbb{E}$, and Section 6 on the nonemptiness and universality problems of these. Section 7 switches the focus to QLAs with word and language aggregators LimInf and LimSup. Section 8 concludes with potential research directions. We summarize our decidability and complexity results in Table 1. We provide proof sketches in the main text; the details are available in the full version.

To present a comprehensive picture, we overcome several technical challenges: First, for the evaluation problem (Section 5), we demonstrate that the value of an ω-regular language L may not be realized (i) by a lasso word in L for limit-average (a.k.a. mean-payoff) QLAs and (ii) by any word in L for discounted-sum QLAs. Although the evaluation problem for these automata is not always solvable, for its solvable cases we resolve these issues by proving that (i) in the limit-average case, the value of L can still be computed by analyzing strongly connected components even if it is not realized by a lasso word, and (ii) in the discounted-sum case, the value of L matches the value of its safety closure L' and is realized

by some word in L'. These results yield PTime algorithms for the evaluation of these QLAs. We complement these results with hardness proofs via reductions from universality problems of the underlying QWAs.

Second, for nonemptiness and universality (Section 6), we examine the behavior of common QWA classes regarding the greatest lower bound of their values – their so-called bottom values [4]. We show (i) discounted-sum QWAs always have a word achieving their bottom value, and (ii) the bottom value of limit-superior-average QWAs can be approximated by lasso words, contrasting sharply with limit-inferior-average QWAs [12]. This enables us to establish the hardness of nonemptiness and universality for certain classes of QLAs. Moreover, we prove that for most classes of QLAs the unrestricted versions of nonemptiness and universality coincide with their finite-state restrictions, indicating a finite-model property.

Third, for QLAs using LimInf and LimSup as word and language aggregators (Section 7), we study values realized by infinitely many runs of a word and infinitely many words of a given language. Using combinatorial arguments, we characterize structural patterns in ω-automata that precisely capture when a value is infinitely realizable with the run aggregators Inf, Sup, LimInf, and LimSup.

Related Work. Our work builds on quantitative languages and quantitative word automata [10, 12], as well as on games with quantitative objectives [14]. There have been other definitions of quantitative hyperproperties [16, 28, 31].

While our motivations align, QLAs in their current form and the formalisms of [16, 28] are orthogonal. In [16, 28], a quantitative hyperproperty is based on counting the number of executions that appear in a given relation. Therefore, these formalisms cannot express, e.g., the worst-case server uptime, which QLAs can (see Section 3). Similarly, QLAs with the aggregators we considered cannot express properties that can only be defined using counting or cardinality constraints.

In [31], the authors provide a predicate-transformer theory for reasoning on finite executions of imperative programs. Their formalism can define quantitative hyperproperties that can be expressed as functions from a set or distribution of values at a program's final state to a single value, which is closer to our view compared to [16,28]. For example, it can express the maximal value of a variable x given the value of variable y upon termination (akin to QLAs) or the variance of a random variable. However, it cannot express long-run aggregates such as limit superior or limit averages. Conversely, QLAs cannot express variances with the current aggregators, since it requires combining two expectations at the language-aggregator level. To the best of our knowledge, our work is the first to define quantitative hyperproperties as functions from sets or distributions of infinite words to quantitative values, and therefore the first to study the specification and verification of such properties through automata.

2 Definitional Framework

Let Σ be a finite *alphabet* of letters. A *word* (or *trace*) over Σ is a finite or infinite sequence of letters from Σ. We denote by $|w|$ the length of a finite word w. We denote by Σ^* (resp. Σ^ω) the set of all finite (resp. infinite) words over Σ. An infinite word w is *ultimately periodic* (a.k.a. *lasso*) iff $w = uv^\omega$ for some $u, v \in \Sigma^*$ such that $|v| \geq 1$. A *language* is a set of infinite words. Given $u \in \Sigma^*$ and $w \in \Sigma^\omega$, we write $u \prec w$ when u is a prefix of w. We denote by $\mathbb{N}$ the set of natural numbers (including 0), $\mathbb{Q}$ the set of rational numbers, $\mathbb{R}$ the set of real numbers. We further let $\overline{\mathbb{N}} = \mathbb{N} \cup \{\infty\}$ and $\overline{\mathbb{R}} = \mathbb{R} \cup \{-\infty, +\infty\}$. Consider a set S. An S-multiset is a function $M : S \to \overline{\mathbb{N}}$ that maps each element of S to a value denoting

its multiplicity. The support of a multiset M is the set $supp(M) = \{x \in S \mid M(x) \geq 1\}$ of distinct elements in M. A *value domain* $\mathbb{D}$ is a nontrivial complete lattice. A *quantitative property* is a total function $\Phi : \Sigma^\omega \to \mathbb{D}$.

Quantitative Word Automata. A *weighted labeled transition system* is a tuple $\mathcal{T} = (\Sigma, Q, s, \delta, d)$, where: Σ is a finite alphabet, Q is a finite nonempty set of states, $s \in Q$ is the initial state, $\delta \colon Q \times \Sigma \to 2^{\mathbb{Q} \times Q}$ is a finite transition function over weight-state pairs, and $d : Q \times \Sigma \times \mathbb{Q} \times Q \to [0, 1]$ is a probability distribution such that for all $q \in Q$ and $a \in \Sigma$, (i) $d(q, a, x, q') > 0$ iff $(x, q') \in \delta(q, a)$, and (ii) $\sum_{(x,q') \in \delta(q,a)} d(q, a, x, q') = 1$. Given a transition system $\mathcal{T}$, the *dual* of $\mathcal{T}$ is a copy of $\mathcal{T}$ with the weights multiplied by -1.

A *transition* is a tuple $(q, \sigma, x, q') \in Q \times \Sigma \times \mathbb{Q} \times Q$ such that $(x, q') \in \delta(q, \sigma)$, denoted $q \xrightarrow{\sigma:x} q'$. Given a transition $t = q \xrightarrow{\sigma:x} q'$, we denote its weight by $\gamma(t) = x$. We say that $\mathcal{T}$ is *complete* (a.k.a. *total*) iff $|\delta(q, a)| \geq 1$ for every $q \in Q$ and $a \in \Sigma$, and *deterministic* iff $|\delta(q, a)| = 1$ for every $q \in Q$ and $a \in \Sigma$. Throughout the paper, we assume that weighted labeled transition systems are complete.

Although weighted labeled transition systems are probabilistic by definition, they can be viewed as *nondeterministic* by considering only the support of d, i.e., treating all transitions with positive probability as nondeterministic transitions.

A *run* of $\mathcal{T}$ is an infinite sequence $\rho = q_0 \xrightarrow{\sigma_0:x_0} q_1 \xrightarrow{\sigma_1:x_1} \dots$ of transitions such that $q_0 = s$ and $(x_i, q_{i+1}) \in \delta(q_i, \sigma_i)$ for all $i \geq 0$. We write $\gamma(\rho)$ for the corresponding infinite sequence $x_0 x_1 \dots$ of rational weights. Given a word $w \in \Sigma^\omega$, we denote by $R(\mathcal{T}, w)$ the set of runs of w over $\mathcal{T}$. Let $\rho = t_0 t_1 \dots$ be a run and $r = t_0 t_1 \dots t_n$ be a finite prefix of ρ. The probability of r is $\prod_{i=0}^{n} d(t_i)$. For each infinite word w, we define $\mu_{\mathcal{T},w}$ as the unique probability measure over Borel sets of infinite runs of $\mathcal{T}$ over w, induced by the transition probabilities d.

A *run aggregator* (a.k.a. value function) is a function $f : \mathbb{Q}^\omega \to \overline{\mathbb{R}}$ that accumulates an infinite sequence of weights into a single value. We consider the run aggregators below over an infinite sequence $x = x_0 x_1 \dots$ of rational weights and a discount factor $\lambda \in \mathbb{Q} \cap (0, 1)$. We write DSum when the discount factor $\lambda \in \mathbb{Q} \cap (0, 1)$ is unspecified. The run aggregators Inf and Sup (resp., LimInf and LimSup, $\mathsf{LimInfAvg}$ and $\mathsf{LimSupAvg}$) are duals. The dual of DSum_λ is itself.

- $\mathsf{Inf}(x) = \inf\{x_n \mid n \in \mathbb{N}\}$
- $\mathsf{Sup}(x) = \sup\{x_n \mid n \in \mathbb{N}\}$
- $\mathsf{LimInf}(x) = \liminf_{n\to\infty} x_n$
- $\mathsf{LimSup}(x) = \limsup_{n\to\infty} x_n$
- $\mathsf{LimInfAvg}(x) = \liminf_{n\to\infty} \left(\frac{1}{n} \sum_{i=0}^{n-1} x_i\right)$
- $\mathsf{LimSupAvg}(x) = \limsup_{n\to\infty} \left(\frac{1}{n} \sum_{i=0}^{n-1} x_i\right)$
- $\mathsf{DSum}_\lambda(x) = \sum_{i\geq 0} \lambda^i x_i$

A *word aggregator* is a function $g : \overline{\mathbb{N}}^{\overline{\mathbb{R}}} \to \overline{\mathbb{R}}$ that accumulates a multiset of values obtained from the runs of a word into a single value. We consider the word aggregators defined below where M is an $\overline{\mathbb{R}}$-multiset and μ is a probability measure over $supp(M)$. The word aggregators Inf and Sup (resp., LimInf and LimSup) are duals. The dual of $\mathbb{E}_\mu$ is itself.

- $\mathsf{Inf}(M) = \inf\{x \mid M(x) \geq 1\}$
- $\mathsf{Sup}(M) = \sup\{x \mid M(x) \geq 1\}$
- $\mathbb{E}_\mu(M) = \int_{x \in supp(M)} x d\mu(x)$
- $\mathsf{LimInf}(M) = \inf\{x \mid M(x) = \infty\}$
- $\mathsf{LimSup}(M) = \sup\{x \mid M(x) = \infty\}$

A *quantitative word automaton* (QWA) [12] is a tuple $\mathcal{A} = (g, f, \mathcal{T})$ where $\mathcal{T}$ is a complete weighted labeled transition system, f is a run aggregator, and g is a word aggregator. Given a word w, a transition system $\mathcal{T}$, and a run aggregator f, we define an $\overline{\mathbb{R}}$-multiset $M_{w,\mathcal{T},f}$ such that for every $x \in \overline{\mathbb{R}}$ the value $M_{w,\mathcal{T},f}(x)$ equals the number of runs ρ of $\mathcal{T}$ over w such that

$f(\gamma(\rho)) = x$. For all $x \in \mathit{supp}(M_{w,\mathcal{T},f})$ we define $\mu(x) = \mu_{\mathcal{T},w}(\{\rho \in R(\mathcal{T}, w) \mid f(\gamma(\rho)) = x\})$. We let $\mathcal{A}(w) = g(M_{w,\mathcal{T},f})$ for all $w \in \Sigma^\omega$ and write $\mathbb{E}$ for $\mathbb{E}_\mu$ when μ is clear from the context.

A QWA is *nondeterministic* when its word aggregator is Sup, and *universal* when Inf. The *top value* of a word automaton $\mathcal{A}$ is $\top_\mathcal{A} = \sup_{w\in\Sigma^\omega} \mathcal{A}(w)$, and its *bottom value* is $\bot_\mathcal{A} = \inf_{w\in\Sigma^\omega} \mathcal{A}(w)$. We say that $\top_\mathcal{A}$ (resp. $\bot_\mathcal{A}$) is *achievable* iff there exists a word w with $\mathcal{A}(w) = \top_\mathcal{A}$ (resp. $\mathcal{A}(w) = \bot_\mathcal{A}$). Given a word automaton $\mathcal{A} = (g, f, \mathcal{T})$ its *dual* is $\hat{\mathcal{A}} = (\hat{g}, \hat{f}, \hat{\mathcal{T}})$ where $\hat{g}$, $\hat{f}$, and $\hat{\mathcal{T}}$ are the duals of g, f, and $\mathcal{T}$, respectively. For a QWA $\mathcal{A} = (g, f, \mathcal{T})$ and a word w, if $M_{w,\mathcal{T},f}(x)$ is finite for all $x \in \overline{\mathbb{R}}$, then we let $\mathcal{A}(w) = \top_\mathcal{A}$ when $g = \mathsf{LimInf}$ and $\mathcal{A}(w) = \bot_\mathcal{A}$ when $g = \mathsf{LimSup}$.

Boolean word automata are a special case of QWAs with weights in $\{0, 1\}$ and $f \in \{\mathsf{Inf}, \mathsf{Sup}, \mathsf{LimInf}, \mathsf{LimSup}\}$. In particular, a nondeterministic Büchi automaton is a boolean word automaton with $g = \mathsf{Sup}$ and $f = \mathsf{LimSup}$. Given a boolean word automaton $\mathcal{B}$, we write $L(\mathcal{B}) = \{w \in \Sigma^\omega \mid \mathcal{B}(w) = 1\}$ for its language.

Quantitative Language Automata. A *language generator* is a function $G : 2^{\Sigma^\omega} \to [0, 1]$. A language generator G is *nonprobabilistic* iff there is a language $L_G \subseteq \Sigma^\omega$ such that $G(L_G) = 1$ and $G(L) = 0$ for every other language $L \neq L_G$. A nonprobabilistic language generator G is *nonempty* iff $L_G \neq \emptyset$. A language generator G is *probabilistic* iff it defines a probability measure μ_G over Σ^ω. A *nonprobabilistic quantitative hyperproperty* (resp. *probabilistic quantitative hyperproperty*) is a total function from the set of all nonprobabilistic (resp. probabilistic) language generators to a value domain $\mathbb{D}$.

A *language aggregator* is a function $h : \overline{\mathbb{N}}^{\overline{\mathbb{R}}} \to \overline{\mathbb{R}}$ that accumulates a multiset of values obtained from a set of words into a single value. We consider the language aggregators Inf, Sup, LimInf, LimSup, and $\mathbb{E}_\mu$, which are defined the same as for the word aggregators above.

A *quantitative language automaton* (QLA) is a pair $\mathbb{A} = (h, \mathcal{A})$ where $\mathcal{A} = (g, f, \mathcal{T})$ is a QWA and h is a language aggregator. Consider a language generator G and a QWA $\mathcal{A}$. If G is nonprobabilistic (resp. probabilistic), we define $M_{G,\mathcal{A}}$ as an $\overline{\mathbb{R}}$-multiset such that for every $x \in \overline{\mathbb{R}}$ the value $M_{G,\mathcal{A}}(x)$ equals the number of words w in $L_G \subseteq \Sigma^\omega$ (resp. in Σ^ω) such that $\mathcal{A}(w) = x$. Moreover, if G is probabilistic, we additionally let $\mu(x) = \mu_G(\{w \in \Sigma^\omega \mid \mathcal{A}(w) = x\})$. Then, we let $\mathbb{A}(G) = h(M_{G,\mathcal{A}})$ for every language generator G where μ is as above. We write $\mathbb{E}$ for $\mathbb{E}_\mu$ when μ is clear from the context.

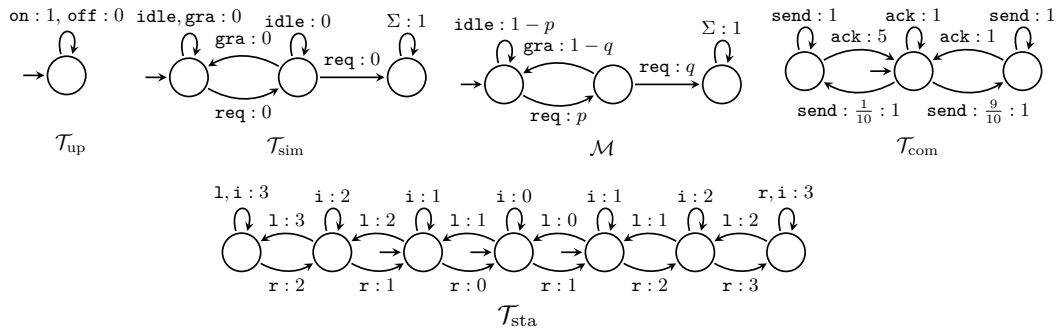
Below, we assume the input to a QLA is a nonprobabilistic (resp. probabilistic) language generator when $h \neq \mathbb{E}$ (resp. $h = \mathbb{E}$). We write L to denote a nonprobabilistic language generator, and μ to denote a probabilistic one.

The *top value* of a QLA $\mathbb{A}$, denoted $\top_\mathbb{A}$, is the supremum of the values $\mathbb{A}(G)$ over all inputs G, and its *bottom value*, denoted $\bot_\mathbb{A}$, is the infimum. For a QLA $\mathbb{A} = (h, \mathcal{A})$ and a nonprobabilistic language generator L, if $M_{L,\mathcal{A}}(x) = 0$ for all $x \in \overline{\mathbb{R}}$ (i.e., when $L = \emptyset$), then we let $\mathbb{A}(L) = \top_\mathbb{A}$ when $h = \mathsf{Inf}$ and $\mathbb{A}(L) = \bot_\mathbb{A}$ when $h = \mathsf{Sup}$. Similarly, if $M_{L,\mathcal{A}}(x)$ is finite for all $x \in \overline{\mathbb{R}}$, then we let $\mathbb{A}(L) = \top_\mathbb{A}$ when $h = \mathsf{LimInf}$ and $\mathbb{A}(L) = \bot_\mathbb{A}$ when $h = \mathsf{LimSup}$.

▶ **Proposition 2.1.** *Consider a QLA $\mathbb{A} = (h, \mathcal{A})$ with $h \in \{\mathsf{Inf}, \mathsf{Sup}, \mathbb{E}\}$. Then, $\top_\mathbb{A} = \top_\mathcal{A}$ and $\bot_\mathbb{A} = \bot_\mathcal{A}$.*

Given a QLA $\mathbb{A} = (h, \mathcal{A})$ its *dual* is $\hat{\mathbb{A}} = (\hat{h}, \hat{\mathcal{A}})$ where $\hat{h}$ and $\hat{\mathcal{A}}$ are the duals of h and $\mathcal{A}$.

▶ **Proposition 2.2.** *Consider a QLA $\mathbb{A}$ and its dual $\hat{\mathbb{A}}$. Then, $\mathbb{A}(G) = -\hat{\mathbb{A}}(G)$ for every language generator G.*



$\mathcal{T}_{\text{up}}$ $\mathcal{T}_{\text{sim}}$ $\mathcal{M}$ $\mathcal{T}_{\text{com}}$ $\mathcal{T}_{\text{sta}}$

Figure 1 Weighted labeled transition systems $\mathcal{T}_{\text{up}}$, $\mathcal{T}_{\text{sim}}$, $\mathcal{T}_{\text{com}}$, and $\mathcal{T}_{\text{sta}}$ can be used to specify QWAs and QLAs with various aggregators, as demonstrated in Section 3. Transitions are marked with $\sigma : p : x$ where σ denotes a letter, p the transition probability (dropped when $p = 1$), and x the transition weight. The Markov chain $\mathcal{M}$ is a language generator where $0 \leq p, q \leq 1$ (its transitions are not weighted, and those with probability 0 are not shown).

3 Applications of Language Automata

While QWAs describe system specifications *per execution*, QLAs describe *system-wide* specifications. We show several applications of QLAs for specifying quantitative hyperproperties. Notice that QLAs are an *operational* specification language – their underlying word automata serve as part of the specification itself.

Server Uptime. QWAs can model performance metrics for individual executions. Let $\Sigma = \{\texttt{on}, \texttt{off}\}$ be a finite alphabet of observations modeling a server's activity. Consider the weighted labeled transition system $\mathcal{T}_{\text{up}}$ given in Figure 1, and let $\mathcal{A}_{\text{up}} = (g, f, \mathcal{T}_{\text{up}})$ be a QWA where $g = \mathsf{Sup}$ and $f = \mathsf{LimInfAvg}$. The automaton $\mathcal{A}_{\text{up}}$ maps each infinite word w to the long-run ratio of $\texttt{on}$, i.e., the average uptime of the execution modeled by w. For example, if $w = \texttt{on} \cdot (\texttt{on} \cdot \texttt{off})^\omega$, we have $\mathcal{A}(w) = 0.5$.

QLAs can model *system-wide* performance metrics. Let $\mathbb{A}_{\text{up}} = (h, \mathcal{A}_{\text{up}})$. Setting $h = \mathsf{Inf}$ tunes $\mathbb{A}_{\text{up}}$ to map any given language L, capturing the implementation of a server, to the worst-case uptime over all the executions allowed by the implementation, i.e., $\mathbb{A}_{\text{up}}(L) = \inf_{w \in L} \mathcal{A}_{\text{up}}(w)$. For example, let $U = \{\texttt{on}^n\texttt{off}^k \mid n > k \geq 0\}$ and consider the language $L = \{u_1 u_2 \ldots \mid \forall i : u_i \in U\}$ of server executions where each block of $\texttt{on}$'s is followed by a strictly shorter block of $\texttt{off}$'s. Although for every $u \in U$ there are strictly more $\texttt{on}$'s than $\texttt{off}$'s, we have $w = s_1 s_2 \ldots \in L$ where $s_i = \texttt{on}^i\texttt{off}^{i-1} \in U$ for each $i \geq 1$, thus $\mathbb{A}_{\text{up}}(L) = 0.5$. In particular, notice that the value of $\mathbb{A}_{\text{up}}(L)$ cannot be achieved by any ultimately periodic word in L; we will later show that this may happen even for ω-regular languages (Proposition 5.2). As another example of using QLAs to specify quantitative hyperproperties, setting $h = \mathsf{LimInf}$ tunes $\mathbb{A}_{\text{up}}$ to express the "almost" worst-case uptime by considering only the uptime values realized by infinitely many executions. For example, let $L = \Sigma^*\texttt{on}^\omega \cup \{(\texttt{on}^n\texttt{off})^\omega \mid n \geq 0\}$. The part $\Sigma^*\texttt{on}^\omega$ yields infinitely many executions with uptime 1, while each $(\texttt{on}^n\texttt{off})^\omega$ with uptime $\frac{n}{n+1}$ appears only once. Thus, even though the infimum over L is 0, only uptime 1 occurs infinitely often, so $\mathbb{A}(L) = 1$.

Implementation Distance. QWAs can specify the distance of individual executions to a desired boolean trace property. Let $\Sigma = \{\texttt{req}, \texttt{gra}, \texttt{idle}\}$ be a finite alphabet of observations modeling a server receiving requests and issuing grants, and consider the boolean safety

property P requiring that no two requests are simultaneously open (i.e., there is a $\texttt{gra}$ between every two $\texttt{req}$'s). Consider the transition system $\mathcal{T}_{\text{sim}}$ given in Figure 1, and let $\mathcal{A}_{\text{sim}} = (g, f, \mathcal{T}_{\text{sim}})$ be a QWA where $f = \mathsf{DSum}_{0.5}$ and $g \in \{\mathsf{Inf}, \mathsf{Sup}\}$ (the choice of g does not matter since $\mathcal{T}_{\text{sim}}$ is deterministic). The automaton $\mathcal{A}_{\text{sim}}$ maps each infinite word w to the smallest Cantor distance between w and some word $w' \in P$, i.e., $\mathcal{A}_{\text{sim}}(w) = \inf_{w' \in P} d(w, w')$. For example, if $w = \texttt{req} \cdot \texttt{idle} \cdot \texttt{req}^\omega$, we have $\mathcal{A}(w) = 0.25$.

QLAs can specify the distance of *systems* to a desired boolean trace property. Let $\mathbb{A}_{\text{sim}} = (h, \mathcal{A}_{\text{sim}})$. Setting $h = \mathsf{Sup}$ tunes $\mathbb{A}_{\text{sim}}$ to map any given language L, representing a server implementation, to the worst-case distance from L to P. In other words, $\mathbb{A}_{\text{sim}}(L) = \sup_{w \in L} \inf_{w' \in P} d(w, w')$ where d denotes the Cantor distance. For example, if $L = \Sigma^* \texttt{req}^2 \Sigma^\omega$, then $\mathbb{A}_{\text{sim}}(L) = 0.5$. Setting $h = \mathsf{Inf}$ tunes $\mathbb{A}_{\text{sim}}$ to map each language to the best-case distance from L to P. For example, if L is as above, the value of L would be 0 although all the words in L violate P. This is because for every $i \geq 0$ the word $w_i = \texttt{idle}^i \texttt{req}^2 \texttt{idle}^\omega$ belongs to L and $\lim_{i \to \infty} \mathbb{A}_{\text{sim}}(w_i) = 0$, highlight a challenge for the evaluation of discounted-sum QLAs: the value of L may not be realized by any word in L, even if L is omega-regular. Setting $h = \mathbb{E}$ tunes $\mathbb{A}_{\text{sim}}$ to map each language to the average-case distance to P. For example, consider the Markov chain $\mathcal{M}$ from Figure 1. The expected value can be computed by solving the corresponding system of linear equations, resulting in $\mathbb{A}_{\text{sim}}(\mu_{\mathcal{M}}) = \frac{2pq}{2+p(1+q)}$.

Robot Stability. QWAs can express stability constraints of individual executions. Let $\Sigma = \{\texttt{l}, \texttt{r}, \texttt{i}\}$ be a finite alphabet representing robot movements – *left*, *right*, and *idle* – on a one-dimensional finite grid. An execution is ε-stable, for some $\varepsilon \geq 0$, if there exists $0 \leq \delta \leq \varepsilon$ such that whenever the system starts within a δ-ball around the origin, it remains indefinitely within an ε-ball around the origin. For a fixed $\delta \geq 0$, QWAs can express the most permissive ε associated with each execution. Consider the weighted labeled transition system $\mathcal{T}_{\text{sta}}$ given in Figure 1, and let $\mathcal{A}_{\text{sta}} = (\mathsf{Sup}, \mathsf{Sup}, \mathcal{T}_{\text{sta}})$. The automaton captures the scenario where $\delta = 1$, meaning the robot starts at most one step away from the origin. Transition weights indicate distances from the origin, and a run's value is the maximal distance reached. Thus, the automaton's value on a word is the worst-case distance (or most permissive ε) over all initial positions. For instance, the word $w = (\texttt{l} \cdot \texttt{r})^\omega$ has three runs with values 1, 1, and 2, so $\mathcal{A}_{\text{sta}}(w) = 2$.

QLAs can express stability constraints of *systems*. Let $\mathbb{A}_{\text{sta}} = (h, \mathcal{A}_{\text{sta}})$. If $h = \mathsf{Sup}$, then given a language L modeling the robot's behavior, the automaton $\mathbb{A}_{\text{sta}}$ maps L to least upper bound of the set of per-execution ε values obtained from L and $\mathcal{A}_{\text{sta}}$. In other words, $\mathbb{A}_{\text{sta}}(L) = \mathsf{Sup}_{w \in L} \mathbb{A}_{\text{sta}}(w)$ is the smallest ε ensuring all executions in L are ε-stable with $\delta = 1$. Alternatively, if $h = \mathsf{LimSup}$, then $\mathbb{A}_{\text{sta}}(L)$ captures the smallest ε such that infinitely many executions are ε-stable for $\delta = 1$, allowing to discard "outlier" values achieved only by finitely many executions.

Communication Channel Cost. Probabilistic QWAs can specify the expected cost of individual executions. Let $\Sigma = \{\texttt{send}, \texttt{ack}\}$ be an alphabet modeling a communication channel. Consider the transition system $\mathcal{T}_{\text{com}}$ given in Figure 1, and let $\mathcal{A}_{\text{com}} = (\mathbb{E}, \mathsf{LimSup}, \mathcal{T}_{\text{com}})$ be a probabilistic QWA. Each run of the automaton is mapped to a long-term maximal cost, i.e., LimSup of the corresponding weight sequence. Then, each infinite word is mapped to the expected value over the distribution of its runs. For example, considering $w = (\texttt{send} \cdot \texttt{ack})^\omega$ gives us $\mathcal{A}_{\text{com}}(w) = 5$ because the set of runs with a LimSup value 5 (i.e., those in which the high-cost cycle occurs infinitely often) has probability 1.

QLAs can specify the aggregate cost of a communication channel. Let $\mathbb{A}_{\text{com}} = (h, \mathcal{A}_{\text{com}})$. If $h = \mathbb{E}$, the QLA $\mathbb{A}_{\text{com}}$ specifies the expected cost of the underlying probabilistic model of a communication channel. Consider a Markov chain defining a uniform probability measure μ over the alphabet $\Sigma = \{\texttt{send}, \texttt{ack}\}$. Then, $\mathbb{A}(\mu) = 1.05$ which can be computed by analyzing the product of the Markov chain defining μ and the underlying word automaton $\mathcal{A}_{\text{com}}$.

4 Problems on Language Automata

Let $\mathbb{A} = (h, \mathcal{A})$ be a QLA. We study the following problems for $\triangleright \in \{>, \geq\}$.

Nonprobabilistic Evaluation ($h \neq \mathbb{E}$). Given a Büchi automaton $\mathcal{B}$, compute $\mathbb{A}(\mathcal{B})$.
Probabilistic Evaluation ($h = \mathbb{E}$). Given a finite-state Markov chain $\mathcal{M}$, compute $\mathbb{A}(\mathcal{M})$.
$\triangleright$-Nonemptiness. Given $k \in \mathbb{Q}$, is $\mathbb{A}(G) \triangleright k$ for some (nonempty) language generator G?
$\triangleright$-Universality. Given $k \in \mathbb{Q}$, is $\mathbb{A}(G) \triangleright k$ for every (nonempty) language generator G?

For nonemptiness and universality, we also consider the following variants with restricted quantification over language generators.

Borel. Nonprobabilistic generators G with L_G Borel in Σ^ω under the Cantor topology.
Markov. Probabilistic generators G with μ_G Markovian.
Finite-state. Nonprobabilistic generators G with L_G ω-regular; probabilistic generators G with μ_G finite-state Markovian.

We establish the relations between the questions about QLAs and the corresponding questions about their underlying QWA. For completeness, we provide these problem definitions here. Consider a QWA $\mathcal{A}$, a rational $k \in \mathbb{Q}$, and $\triangleright \in \{>, \geq\}$. Then, $\mathcal{A}$ is $\triangleright$-nonempty (resp -universal) for k iff $\mathcal{A}(w) \triangleright k$ for some (resp. all) $w \in \Sigma^\omega$. Similarly as for the finite-state restriction for QLAs problems, the *lasso-word restriction* for these problems requires quantifying over lasso words instead of all words. Moreover, $\mathcal{A}$ is *approximate-nonempty* for k iff $\top_{\mathcal{A}} \geq k$, i.e., if there exists $w \in \Sigma^\omega$ such that $\mathcal{A}(w) \geq k$ or if for every $\varepsilon > 0$ there exists w such that $\mathcal{A}(w) > k - \varepsilon$. Hence, the achievability of $\top_{\mathcal{A}}$ implies that approximate-nonemptiness and $\geq$-nonemptiness are equivalent problems. Note that $\top_{\mathcal{A}} > k$ iff $\mathcal{A}(w) > k$ for some word w – it holds independently of the achievability of $\top_{\mathcal{A}}$. Dually, $\mathcal{A}$ is *approximate-universal* for k iff $\bot_{\mathcal{A}} > k$.

For the classes of QLAs we consider, the Borel (resp. Markov) restrictions of the decision problems coincide with the unrestricted cases. Thus, for their decision problems we only focus on the unrestricted and finite-state cases.

▶ **Proposition 4.1.** *Consider a QLA $\mathbb{A} = (h, \mathcal{A})$ with $h \in \{\mathsf{Inf}, \mathsf{Sup}, \mathsf{LimSup}, \mathsf{LimInf}\}$ (resp. $h = \mathbb{E}$), a rational $k \in \mathbb{Q}$, and $\triangleright \in \{>, \geq\}$. Then, $\mathbb{A}$ is $\triangleright$-nonempty for k iff $\mathbb{A}$ is Borel $\triangleright$-nonempty (resp. Markov $\triangleright$-nonempty) for k. The same holds for universality.*

For QLAs with language aggregators Inf, Sup, and $\mathbb{E}$, we demonstrate that their decision problems often reduce to those of their underlying QWAs. Moreover, we identify several cases in which QLAs enjoy a finite-model property.

▶ **Proposition 4.2.** *Consider a QLA $\mathbb{A} = (h, \mathcal{A})$ with $h \in \{\mathsf{Inf}, \mathsf{Sup}, \mathbb{E}\}$, a rational $k \in \mathbb{Q}$, and $\triangleright \in \{>, \geq\}$.*

(a) *Unrestricted variant*
  (i) *If $h \neq \mathsf{Sup}$ or $\triangleright = >$, then $\mathbb{A}$ is $\triangleright$-nonempty for k iff $\mathcal{A}$ is $\triangleright$-nonempty for k.*
  (ii) *If $h = \mathsf{Sup}$, then $\mathbb{A}$ is $\geq$-nonempty for k iff $\mathcal{A}$ is approximate-nonempty for k.*
(b) *Finite-state variant*
  (i) *If $h = \mathsf{Sup}$, then $\mathbb{A}$ is finite-state $\triangleright$-nonempty for k iff $\mathbb{A}$ is $\triangleright$-nonempty for k.*

(ii) *If $h = \mathsf{Inf}$, then $\mathbb{A}$ is finite-state $\triangleright$-nonempty for k iff $\mathcal{A}$ is lasso-word $\triangleright$-nonempty for k.*
(iii) *If $\top_{\mathcal{A}}$ is achievable by a lasso word, then $\mathbb{A}$ is finite-state $\triangleright$-nonempty for k iff $\mathbb{A}$ is $\triangleright$-nonempty for k iff $\mathcal{A}$ is lasso-word $\triangleright$-nonempty for k.*

For universality, we have the duals where we exchange Inf *with* Sup, $>$ *with* $\geq$, *nonempty with universal, approximate-nonempty with approximate-universal, and* $\top_{\mathcal{A}}$ *with* $\bot_{\mathcal{A}}$.

We show that approximating top or bottom values via lasso words lets us reduce certain unrestricted variants to their finite-state counterparts, establishing a finite-model property.

▶ **Proposition 4.3.** *Consider a QLA $\mathbb{A} = (h, \mathcal{A})$ with $h \in \{\mathsf{Inf}, \mathsf{Sup}, \mathbb{E}\}$ and $k \in \mathbb{Q}$. If for every $\varepsilon > 0$ there is a lasso word w such that $\mathcal{A}(w) \geq \top_{\mathcal{A}} - \varepsilon$, then $\mathbb{A}$ is $>$-nonempty for k iff $\mathbb{A}$ is finite-state $>$-nonempty for k. Dually, if for every $\varepsilon > 0$ there is a lasso word w such that $\mathcal{A}(w) \leq \bot_{\mathcal{A}} + \varepsilon$, then $\mathbb{A}$ is $\geq$-universal for k iff $\mathbb{A}$ is finite-state $\geq$-universal for k.*

5 Solving Evaluation

In this section, we investigate the evaluation problem for QLAs with language and word aggregators $h, g \in \{\mathsf{Inf}, \mathsf{Sup}, \mathbb{E}\}$, for which we provide a full picture of complexity results. First, in Section 5.1, we focus on the nonprobabilistic evaluation problem (where $h \in \{\mathsf{Inf}, \mathsf{Sup}\}$) and then, in Section 5.2, on the probabilistic one (where $h = \mathbb{E}$).

5.1 Nonprobabilistic Evaluation

First, we consider the evaluation problem for QLAs with $h \in \{\mathsf{Inf}, \mathsf{Sup}\}$. We start with QLAs whose underlying word automata are universal or nondeterministic (i.e., $g \in \{\mathsf{Inf}, \mathsf{Sup}\}$). We study various run aggregators f separately and show that the problem is in PTime when the word aggregator g and the language aggregator h coincide. When they differ, the problem becomes harder: while it remains algorithmically solvable in PSpace for the "standard" run aggregators (i.e., $f \in \{\mathsf{Inf}, \mathsf{Sup}, \mathsf{LimInf}, \mathsf{LimSup}\}$), we show that it is not computable for limit-average and at least as hard as a long-standing open problem for discounted-sum. Finally, for QLAs whose underlying word automata are probabilistic (i.e., $g = \mathbb{E}$), we show that the problem is not computable. At their core, the easiness results rely on analyzing the extreme values of the underlying word automata. Similarly, we establish the hardness results by reductions from word automata problems.

QLAs with Standard QWAs. For QLAs with run aggregators $f \in \{\mathsf{Inf}, \mathsf{Sup}, \mathsf{LimInf}, \mathsf{LimSup}\}$, the nonprobabilistic evaluation problem can be solved by reasoning on lasso words since both top and bottom values of the underlying word automata are realized by lasso words.

▶ **Theorem 5.1.** *Consider a QLA $\mathbb{A} = (h, (g, f, \mathcal{T}))$ with $f \in \{\mathsf{Inf}, \mathsf{Sup}, \mathsf{LimInf}, \mathsf{LimSup}\}$ and $g, h \in \{\mathsf{Inf}, \mathsf{Sup}\}$. Let $S \subseteq \Sigma^\omega$ be an ω-regular language given by a Büchi automaton. The value $\mathbb{A}(S)$ is computable in PTime when $g = h$ and in PSpace when $g \neq h$.*

QLAs with Limit-Average QWAs. QLAs with run aggregators $f \in \{\mathsf{LimInfAvg}, \mathsf{LimSupAvg}\}$ differ from the previous case in the sense that it is not sufficient to consider only the lasso words in a given ω-regular language, even when the underlying word automaton is deterministic. To witness this, consider the best-case average uptime QLA $\mathbb{A} = (\mathsf{Sup}, (\mathsf{Sup}, \mathsf{LimInfAvg}, \mathcal{T}_{\text{up}}))$ as in Section 3 and Figure 1. Having $L = (\Sigma^* \texttt{off})^\omega$, we get $\mathbb{A}(L) = 1$ as there is a word in L with infinitely many `off`'s but longer and longer blocks of `on`'s, but no lasso word in L has an average uptime of 1.

▶ **Proposition 5.2.** *There is a QLA $\mathbb{A}$ with a run aggregator $f \in \{\mathsf{LimInfAvg}, \mathsf{LimSupAvg}\}$ and an ω-regular language S such that no lasso word in S achieves the value $\mathbb{A}(S)$.*

Nonetheless, for QLAs with matching word and language aggregators, we show that the value of an ω-regular language given by a Büchi automaton $\mathcal{B}$ is computable by analyzing the strongly connected components (SCCs) of the underlying word automaton's product with $\mathcal{B}$ as follows: Among all SCCs that are reachable from the initial state, we find the ones that contain at least one state whose Büchi component is accepting. Then, in each such SCC, we compute the maximum mean weight of its cycles by Karp's algorithm [22]. The largest among these mean values is exactly the value of the given language. Even though such a cycle may not involve an accepting state of $\mathcal{B}$, we can construct a run of the product that visits an accepting state infinitely often while going over this cycle with increasing frequency (hence the long-run average converges to the cycle's mean). When these aggregators differ, the problem is undecidable by reduction from the universality of limit-average QWAs [9, 14, 20].

▶ **Theorem 5.3.** *Consider a QLA $\mathbb{A} = (h, (g, f, \mathcal{T}))$ with $f \in \{\mathsf{LimInfAvg}, \mathsf{LimSupAvg}\}$ and $g, h \in \{\mathsf{Inf}, \mathsf{Sup}\}$. Let $S \subseteq \Sigma^\omega$ be an ω-regular language given by a Büchi automaton. The value $\mathbb{A}(S)$ is computable in PTime when $g = h$ and not computable when $g \neq h$.*

QLAs with Discounted-Sum QWAs. QLAs with the run aggregator $f = \mathsf{DSum}$ have the particular behavior that the value assigned to an ω-regular language L may be not achievable by any word in L, even when the underlying word automaton is deterministic. To witness this, consider $\mathbb{A} = (\mathsf{Sup}, \mathcal{A})$ with $\mathcal{A} = (\mathsf{Sup}, \mathsf{DSum}_{0.5}, \mathcal{T}_{\text{up}})$ as in Figure 1. We have $\mathbb{A}((\Sigma^*\texttt{off})^\omega) = 2$ since $\lim_{n\to\infty} \mathcal{A}(\texttt{on}^n\texttt{off}^\omega) = 2$. However, only for $w = \texttt{on}^\omega \notin (\Sigma^*\texttt{off})^\omega$ we have $\mathcal{A}(w) = 2$.

▶ **Proposition 5.4.** *There is a QLA $\mathbb{A}$ with the run aggregator $f = \mathsf{DSum}$ and an ω-regular language S such that no word in S achieves the value $\mathbb{A}(S)$.*

We establish that such a behavior is not possible when the input language includes all its limit points, i.e., it is safe in the boolean sense [2, 24]: Consider a sequence of words in the safety language whose values approach the supremum. We build by a diagonalization argument an infinite word w whose every finite prefix already appears in the language, so w is in the safety language. Applying the same construction to the corresponding optimal runs yields an infinite run on w. This run's value equals the supremum since the contribution of the remaining tail is bounded by a vanishing geometric series due to discounting.

▶ **Proposition 5.5.** *Consider a QLA $\mathbb{A} = (\mathsf{Sup}, (\mathsf{Sup}, \mathsf{DSum}, \mathcal{T}))$. For every nonempty safety language $S \subseteq \Sigma^\omega$, the value $\mathbb{A}(S)$ is achievable by some run of a word in S.*

The value of a language S matches that of its safety closure S' (i.e., the smallest safety language containing it) because every word in the safety closure can be approximated arbitrarily closely by words from the original language: If S' achieves a value on a word w, we can isolate a prefix of w whose contribution is close enough to the value of w. By construction, the same prefix also occurs in a word of S, and completing the run along this word in S can change the total value by at most an arbitrarily small amount due to discounting. Hence the maximal value in S' can be approximated arbitrarily closely by words in S, and the two suprema coincide.

▶ **Proposition 5.6.** *Consider a QLA $\mathbb{A} = (\mathsf{Sup}, (\mathsf{Sup}, \mathsf{DSum}, \mathcal{T}))$. For every language $S \subseteq \Sigma^\omega$ we have $\mathbb{A}(S) = \mathbb{A}(S')$ where S' is the safety closure of S.*

The observation above helps us provide a PTime algorithm when the word and language aggregators match: We first construct the Büchi automaton's safety closure, so the optimal value is achieved by a run that never reaches the rejecting sink. Then, we compute the product of the underlying word automaton and the safety closure automaton. Computing the best (or worst) discounted sum over all sink-avoiding paths in the product can be done by solving a one-player discounted-payoff game [3]. When the two aggregators differ, the evaluation problem is at least as hard as the universality problem for nondeterministic discounted-sum automata, which is a long-standing open problem [6, 12].

▶ **Theorem 5.7.** *Consider a QLA $\mathbb{A} = (h, (g, \mathsf{DSum}, \mathcal{T}))$ with $g, h \in \{\mathsf{Inf}, \mathsf{Sup}\}$. Let $S \subseteq \Sigma^\omega$ be an ω-regular language given by a Büchi automaton. The value $\mathbb{A}(S)$ is computable in PTime when $g = h$. If $\mathbb{A}(S)$ is computable when $g \neq h$, then the $\geq$-universality of nondeterministic discounted-sum word automata is decidable.*

QLAs with Probabilistic QWAs. When the underlying word automaton is probabilistic, i.e., has the word aggregator $g = \mathbb{E}$, the nonprobabilistic evaluation problem has no algorithmic solution due to inapproximability of their top values [26].

▶ **Theorem 5.8.** *Consider a QLA $\mathbb{A} = (h, (\mathbb{E}, f, \mathcal{T}))$ with $f \in \{\mathsf{Inf}, \mathsf{Sup}, \mathsf{LimInf}, \mathsf{LimSup}, \mathsf{LimInfAvg}, \mathsf{LimSupAvg}, \mathsf{DSum}\}$ and $h \in \{\mathsf{Inf}, \mathsf{Sup}\}$. Let μ be a probability measure given by a finite-state Markov chain. The value $\mathbb{A}(\mu)$ is not computable.*

5.2 Probabilistic Evaluation

Now, we consider the evaluation problem for QLAs with $h = \mathbb{E}$ and follow the same structure as in Section 5.1: we start with the cases of $g \in \{\mathsf{Inf}, \mathsf{Sup}\}$ and study various run aggregators f separately, and then look at the case of $g = \mathbb{E}$.

QLAs with Standard QWAs. First, we provide an ExpTime algorithm for QLAs with the run aggregators $f \in \{\mathsf{Inf}, \mathsf{Sup}, \mathsf{LimInf}, \mathsf{LimSup}\}$. Our proof builds on that of [27, Thm. 8], which considers only $f \in \{\mathsf{Inf}, \mathsf{Sup}\}$. The idea is to determinize the underlying word automaton, which leads to an exponential blow-up, and evaluate its product with the given Markov chain.

▶ **Theorem 5.9.** *Consider a QLA $\mathbb{A} = (\mathbb{E}, (g, f, \mathcal{T}))$ with $f \in \{\mathsf{Inf}, \mathsf{Sup}, \mathsf{LimInf}, \mathsf{LimSup}\}$ and $g \in \{\mathsf{Inf}, \mathsf{Sup}\}$. Let μ be a probability measure given by a finite-state Markov chain. The value $\mathbb{A}(\mu)$ is computable in ExpTime.*

QLAs with Limit-Average QWAs. Undecidability of probabilistic evaluation for limit-average QLAs was shown in [27] by a reduction from the universality problem of quantitative automata on finite words with the summation run aggregator and weights in $\{-1, 0, 1\}$, a.k.a. weighted automata over the tropical semiring of integers [1, 23].

▶ **Theorem 5.10** ([27, Thm. 7]). *Consider a QLA $\mathbb{A} = (\mathbb{E}, (g, f, \mathcal{T}))$ with $f \in \{\mathsf{LimInfAvg}, \mathsf{LimSupAvg}\}$ and $g \in \{\mathsf{Inf}, \mathsf{Sup}\}$. Let μ be a probability measure given by a finite-state Markov chain. The value $\mathbb{A}(\mu)$ is not computable.*

QLAs with Discounted-Sum QWAs. Next, we show the hardness of probabilistic evaluation for DSum QLAs. As in the nonprobabilistic case, we provide a reduction from the universality problem of nondeterministic discounted-sum QWAs.

▶ **Theorem 5.11.** *Consider a QLA* $\mathbb{A} = (\mathbb{E}, (g, \mathsf{DSum}, \mathcal{T}))$ *with* $g \in \{\mathsf{Inf}, \mathsf{Sup}\}$*. Let* μ *be a probability measure given by a finite-state Markov chain. If* $\mathbb{A}(\mu)$ *is computable, then the* $\geq$*-universality of nondeterministic discounted-sum word automata is decidable.*

QLAs with Probabilistic QWAs. Finally, for QLAs with $h = g = \mathbb{E}$, the evaluation problem reduces to evaluating a Markov chain with rewards on infinite words, which is solved using linear programming [21].

▶ **Theorem 5.12** ([21])**.** *Consider a QLA* $\mathbb{A} = (\mathbb{E}, (\mathbb{E}, f, \mathcal{T}))$ *with* $f \in \{\mathsf{Inf}, \mathsf{Sup}, \mathsf{LimInf}, \mathsf{LimSup}, \mathsf{LimInfAvg}, \mathsf{LimSupAvg}, \mathsf{DSum}\}$*. Let* μ *be a probability measure given by a finite-state Markov chain. The value* $\mathbb{A}(\mu)$ *is computable in* PTime*.*

6 Deciding Nonemptiness and Universality

In this section, we investigate the nonemptiness and universality problems for QLAs with language and word aggregators $h, g \in \{\mathsf{Inf}, \mathsf{Sup}, \mathbb{E}\}$. We give a complete picture of decidability results for the unrestricted cases. When the unrestricted cases are decidable, our algorithms provide a solution for the finite-state cases in the same complexity class. When they are undecidable or not known to be decidable, some finite-state cases remain open, which we make explicit in the corresponding statements.

Thanks to the duality between nondeterministic and universal automata, solving the nonemptiness problem solves the universality problem as well.

▶ Remark 6.1. Consider a QLA $\mathbb{A} = (h, \mathcal{A})$ and its dual $\hat{\mathbb{A}}$. Let $k \in \mathbb{Q}$. Thanks to Proposition 2.2, the QLA $\mathbb{A}$ is $\geq$-nonempty (resp. $\geq$-universal) for k iff $\hat{\mathbb{A}}$ is not $>$-universal (resp. $>$-nonempty) for $-k$. The statement holds also for the finite-state restriction.

The rest of the section focuses on the nonemptiness problem and is organized by the type of the underlying QWAs. We first consider the case of nondeterministic automata (i.e., $g = \mathsf{Sup}$) and show that the problem is decidable in PTime for these. Then, we consider universal automata (i.e., $g = \mathsf{Inf}$) with various run aggregators f separately. Similarly as for the evaluation problem, nonemptiness is in PSpace for the "standard" run aggregators, undecidable for limit-average, and at least as hard as a long-standing open problem for discounted sum. Finally, we consider probabilistic automata (i.e., $g = \mathbb{E}$) and show that the problem is undecidable.

QLAs with Nondeterministic QWAs. Let us first consider QLAs whose underlying word automata are nondeterministic, i.e., those where the word aggregator is $g = \mathsf{Sup}$. We show that the nonemptiness problems for such QLAs can be solved efficiently, independently of the choice of the remaining aggregators or additional restrictions on the problem. Intuitively, this is because their top values coincide with those of the underlying QWAs (Proposition 2.1), which are achievable by lasso words and can be computed efficiently.

▶ **Theorem 6.2.** *Consider a QLA* $\mathbb{A} = (h, (\mathsf{Sup}, f, \mathcal{T}))$ *with* $f \in \{\mathsf{Inf}, \mathsf{Sup}, \mathsf{LimInf}, \mathsf{LimSup}, \mathsf{LimInfAvg}, \mathsf{LimSupAvg}, \mathsf{DSum}\}$ *and* $h \in \{\mathsf{Inf}, \mathsf{Sup}, \mathbb{E}\}$*. Let* $\triangleright \in \{>, \geq\}$*. The* $\triangleright$*-nonemptiness of* $\mathbb{A}$ *is in* PTime*. The statement holds also for the finite-state restriction.*

QLAs with Universal Standard QWAs. We turn our attention to QLAs whose underlying word automata are universal, i.e., those where the word aggregator is $g = \mathsf{Inf}$. For run aggregators $f \in \{\mathsf{Inf}, \mathsf{Sup}, \mathsf{LimInf}, \mathsf{LimSup}\}$, we show that computing the automaton's top value suffices, which can be done in PSpace when $g = \mathsf{Inf}$.

▶ **Theorem 6.3.** *Consider a QLA* $\mathbb{A} = (h, (\mathsf{Inf}, f, \mathcal{T}))$ *with* $f \in \{\mathsf{Inf}, \mathsf{Sup}, \mathsf{LimInf}, \mathsf{LimSup}\}$ *and* $h \in \{\mathsf{Inf}, \mathsf{Sup}, \mathbb{E}\}$*. Let* $\triangleright \in \{>, \geq\}$*. The* $\triangleright$*-nonemptiness of* $\mathbb{A}$ *is in* PSpace*. The statement holds also for the finite-state restriction.*

QLAs with Universal Limit-Average QWAs. Next, we focus on QLAs with the limit-average run aggregators. We first show that the bottom value of nondeterministic LimSupAvg QWAs (dually, top value of universal LimInfAvg QWAs) can be approximated arbitrarily closely by lasso words. This is in stark contrast with nondeterministic LimInfAvg QWAs, where an automaton may be lasso-word universal for some threshold k while there exist non-lasso words with values below k [12, Lem. 4]. The intuition behind this is that for LimSupAvg, if $\mathcal{A}(w) = k$ for some word w and value k, then for any $\varepsilon > 0$, there exists a length ℓ such that the average value of any run prefix on w of length at least ℓ is below $k + \varepsilon$, which does not hold in general for LimInfAvg. Consequently, we can find two distant-enough occurrences of the same state and pump the low-average segment between them to obtain a lasso whose overall value stays under the threshold.

▶ **Proposition 6.4.** *Consider a QWA* $\mathcal{A} = (\mathsf{Sup}, \mathsf{LimSupAvg}, \mathcal{T})$*. For every* $\varepsilon > 0$ *there is a lasso word* w *such that* $\mathcal{A}(w) < \bot_{\mathcal{A}} + \varepsilon$*.*

In general, all variants of the nonemptiness problem for universal limit-average QWAs (dually, universality for the nondeterministic ones) are undecidable [9, 14, 20]. Combining these with our approximability result above, we obtain the following.

▶ **Theorem 6.5.** *Consider a QLA* $\mathbb{A} = (h, (\mathsf{Inf}, f, \mathcal{T}))$ *with* $f \in \{\mathsf{LimInfAvg}, \mathsf{LimSupAvg}\}$ *and* $h \in \{\mathsf{Inf}, \mathsf{Sup}, \mathbb{E}\}$*. Let* $\triangleright \in \{>, \geq\}$*. The* $\triangleright$*-nonemptiness of* $\mathbb{A}$ *is undecidable. The statement holds also for the finite-state restriction when (i)* $h \neq \mathbb{E}$ *or (ii)* $f = \mathsf{LimInfAvg}$ *and* $\triangleright = >$*.*

QLAs with Universal Discounted-Sum QWAs. Now, we consider QLAs with the discounted-sum run aggregators. We show that the bottom value of nondeterministic DSum QWAs (dually, top value of universal DSum QWAs) is achievable. At a high level, we repeatedly extend a finite prefix with a letter that does not increase the current infimum over continuation values. In the limit, this process results in an infinite word whose sequence of prefix infima converges to the automaton's bottom value. Since discounted-sum automata are co-safe in the quantitative sense [4, 5, 18], i.e., the value of every infinite word equals the limit of its prefix infima, this constructed word attains exactly the bottom value.

▶ **Proposition 6.6.** *Consider a QWA* $\mathcal{A} = (\mathsf{Sup}, \mathsf{DSum}, \mathcal{T})$*. There is a word* w *such that* $\mathcal{A}(w) = \bot_{\mathcal{A}}$*.*

By combining the achievability result above with Proposition 4.2, we obtain a concise proof to establish that the nonemptiness problem for QLAs whose underlying QWAs are universal DSum QWAs is at least as hard as the universality problem for nondeterministic DSum QWAs, which is a long-standing open problem [6, 12].

▶ **Theorem 6.7.** *Consider a QLA* $\mathbb{A} = (h, (\mathsf{Inf}, \mathsf{DSum}, \mathcal{T}))$ *with* $h \in \{\mathsf{Inf}, \mathsf{Sup}, \mathbb{E}\}$*. Let* $\triangleright \in \{>, \geq\}$*. If the* $\triangleright$*-nonemptiness of* $\mathbb{A}$ *is decidable, then the universality of nondeterministic discounted-sum word automata is decidable. The statement holds also for the finite-state restriction when (i)* $h = \mathsf{Sup}$ *or (ii)* $\triangleright = >$*.*

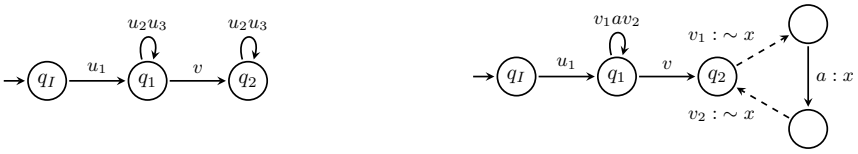


Figure 2 (a) Pattern parameterized by q_2 and n where $u_1, u_2, u_3, v \in \Sigma^+$ are such that $|u_2| = |v|$, $u_2 \neq v$, and $m|v| = |u_2u_3|$ for some $m \in \{1, \ldots, 4n^4\}$. **(b)** Pattern parameterized by $x \in \mathbb{Q}$, $n \in \mathbb{N}$, $\sim \in \{\leq, \geq\}$, and $w \in \Sigma^\omega$, where $u_1, v, v_1, v_2 \in \Sigma^*$ and $a \in \Sigma$ are such that $q_1 \neq q_2$, $w = u(v_1av_2)^\omega$, and $(v)^m = v_1av_2$ for some $m \in \{1, \ldots, 4n^4\}$. Dashed runs visit weights $y \in \mathbb{Q}$ satisfying $y \sim x$.

QLAs with Probabilistic QWAs. Let us now focus on QLAs whose underlying QWA is probabilistic, i.e., where $g = \mathbb{E}$. We show that several variants of the nonemptiness problem for probabilistic QWAs are undecidable by building on the undecidability results of probabilistic word automata [26].

▶ **Theorem 6.8.** *Consider a QWA $\mathcal{A} = (\mathbb{E}, f, \mathcal{T})$ with $f \in \{\mathsf{Inf}, \mathsf{Sup}, \mathsf{LimInf}, \mathsf{LimSup}, \mathsf{LimInfAvg}, \mathsf{LimSupAvg}, \mathsf{DSum}\}$. Let $\triangleright \in \{>, \geq\}$. The $\triangleright$-nonemptiness of $\mathcal{A}$ is undecidable, whether we consider all words from Σ^ω or only lasso words. Moreover, its approximate-nonemptiness is also undecidable.*

Finally, putting together the above undecidability result with Propositions 4.2 and 4.3, we show the undecidability of the nonemptiness problem for QLAs whose underlying QWA is probabilistic.

▶ **Theorem 6.9.** *Consider a QLA $\mathbb{A} = (h, (\mathbb{E}, f, \mathcal{T}))$ with $f \in \{\mathsf{Inf}, \mathsf{Sup}, \mathsf{LimInf}, \mathsf{LimSup}, \mathsf{LimInfAvg}, \mathsf{LimSupAvg}, \mathsf{DSum}\}$ and $h \in \{\mathsf{Inf}, \mathsf{Sup}, \mathbb{E}\}$. Let $\triangleright \in \{>, \geq\}$. The $\triangleright$-nonemptiness of $\mathbb{A}$ is undecidable. The statement holds also for the finite-state restriction when (i) $h \neq \mathbb{E}$ or (ii) $f = \mathsf{DSum}$ and $\triangleright = >$.*

7 Language Automata with Limit Aggregators

We finally focus on QLAs with the aggregators $h, g, f \in \{\mathsf{Inf}, \mathsf{Sup}, \mathsf{LimInf}, \mathsf{LimSup}\}$ where at least one of the language aggregator h or the word aggregator g belongs to $\{\mathsf{LimInf}, \mathsf{LimSup}\}$.

7.1 Deciding Infinite Achievability

To work with QLAs that use LimInf or LimSup as word or language aggregators, we need to reason about the values that appear infinitely often in a multiset. In this subsection, we develop the necessary tools for this purpose, focusing on QLAs with run aggregators $f \in \{\mathsf{Inf}, \mathsf{Sup}, \mathsf{LimInf}, \mathsf{LimSup}\}$.

We begin by characterizing the conditions under which a state in an automaton can be visited infinitely often by infinitely many words. The following lemma uses proof techniques similar to those used in the characterization of ambiguity in finite-state automata [25, 30].

▶ **Lemma 7.1.** *Let $\mathcal{T}$ be a (weighted) labeled transition system with n states and q_I its initial state. For every state q_2 of $\mathcal{T}$, there exist infinitely many words with a run from q_I visiting q_2 infinitely often iff $\mathcal{T}$ complies with the pattern parameterized by q_2 and n given in Figure 2a.*

It is well known that Büchi automata can be effectively complemented in ExpTime [17,29]. Moreover, checking whether a finite-state automaton satisfies a given syntactic pattern, such as the one in Figure 2a, is in NLogSpace [15]. As a direct consequence of Lemma 7.1, keeping a symbolic representation of the complement provides the following PSpace procedure.

▶ **Corollary 7.2.** *Let $\mathcal{B}_1$ and $\mathcal{B}_2$ be two Büchi automata. We can decide in PSpace whether $L(\mathcal{B}_1) \setminus L(\mathcal{B}_2)$ is infinite.*

By interpreting the transitions of a run as input letters, we can use Figure 2a to reason about runs instead of words. Building on Lemma 7.1, we present a similar pattern that incorporates weights, enabling us to identify the words for which an automaton admits infinitely many runs of a given value.

▶ **Lemma 7.3.** *Consider a QWA $\mathcal{A} = (g, f, \mathcal{T})$ with n states, $f \in \{\mathsf{LimInf}, \mathsf{LimSup}\}$, $g \in \{\mathsf{Inf}, \mathsf{Sup}, \mathsf{LimInf}, \mathsf{LimSup}\}$, and initial state q_I. Let $\sim = \leq$ if $f = \mathsf{LimSup}$ and $\sim = \geq$ if $f = \mathsf{LimInf}$. For every $x \in \mathbb{Q}$ and every lasso word $w \in \Sigma^\omega$, we have that w admits infinitely many runs of value x in $\mathcal{A}$ iff $\mathcal{A}$ complies with the pattern in Figure 2b parameterized by x, n, $\sim$, and w.*

As a corollary of Lemma 7.3, we can construct a boolean automaton accepting every lasso word with infinitely many runs of value x in $\mathcal{A}$. To achieve this, we construct a Büchi or co-Büchi automaton that initially guesses a fixed pair of states q_1 and q_2 with the same properties as in Figure 2b. The automaton then guesses on-the-fly the segments u_1 and v from Figure 2b. During the processing of each period v, it simulates $3 \times m$ runs, with $m \in \{1, \ldots, 4n^4\}$, simultaneously verifying weights and connectivity. At the end of each period, the automaton resets (except for the fixed q_1 and q_2) to guess another period and visit an accepting state. Consequently, any accepted lasso word resets infinitely often and ultimately repeats a finite periodic segment v, complying with Figure 2b. Conversely, every word conforming to Figure 2b is accepted by appropriately selecting $v = v_1 a v_2$.

▶ **Corollary 7.4.** *Let $\mathcal{A} = (\mathsf{Sup}, f, \mathcal{T})$ be QWA with $f = \mathsf{LimSup}$ (resp. $f = \mathsf{LimInf}$). Given $x \in \mathbb{Q}$, we can construct in PTime a Büchi automaton (resp. a co-Büchi automaton) recognizing all lasso words admitting infinitely many runs of value x in $\mathcal{A}$.*

7.2 Expressive Power of Limit Aggregators

This subsection investigates the expressive power of word and language aggregators $g, h \in \{\mathsf{Inf}, \mathsf{Sup}, \mathsf{LimInf}, \mathsf{LimSup}\}$. The following proposition generalizes the results of [4, 12] to QLAs.

▶ **Proposition 7.5.** *Consider a QLA $\mathbb{A} = (h, (g, f, \mathcal{T}))$ with $f \in \{\mathsf{Inf}, \mathsf{Sup}\}$, $g \in \{\mathsf{Inf}, \mathsf{Sup}, \mathsf{LimInf}, \mathsf{LimSup}\}$, and h any language aggregator function. We can construct in PTime a QLA $\mathbb{B} = (h, (g, f', \mathcal{T}'))$ with $f' \in \{\mathsf{LimInf}, \mathsf{LimSup}\}$ such that $\mathbb{A}(S) = \mathbb{B}(S)$ for all $S \subseteq \Sigma^\omega$.*

▶ Remark 7.6. The construction of [12] allowing to convert a nondeterministic QWA with run aggregator LimInf to one with LimSup can be extended to QLAs with word aggregator Sup but not with LimSup. By duality, the conversion of a universal LimSup QWA to a LimInf one can be extended to QLAs with word aggregators Inf but not LimInf. This is because deterministic LimInf QWA and deterministic LimSup QWA are expressively incomparable [12]. Hence, there is no conversion that preserves, for every word w, the number runs with the same value over w.

We now focus on word aggregators, first showing that Inf and Sup are at least as expressive as LimInf and LimSup when they are combined with the run aggregators $f \in \{\mathsf{Inf}, \mathsf{Sup}, \mathsf{LimInf}, \mathsf{LimSup}\}$. The construction relies on Corollary 7.4, the closure of QWAs under max and min operation [11], and an argument that equivalence of QWAs with run and word aggregators in {Inf, Sup, LimInf, LimSup} can be determined by considering only lasso words.

▶ **Lemma 7.7.** *Let $\mathbb{A} = (h, (g, f, \mathcal{T}))$ be a QLA with $f \in \{\mathsf{Inf}, \mathsf{Sup}, \mathsf{LimInf}, \mathsf{LimSup}\}$, $g = \mathsf{LimSup}$ (resp. $g = \mathsf{LimInf}$), and h any language aggregator function. We can construct in PSpace a QLA $\mathbb{B} = (h, (g', f', \mathcal{T}'))$ with $f' = \mathsf{LimSup}$ and $g' = \mathsf{Sup}$ (resp. $f' = \mathsf{LimInf}$ and $g' = \mathsf{Inf}$) such that $\mathbb{A}(S) = \mathbb{B}(S)$ for all $S \subseteq \Sigma^\omega$.*

Now, we prove the other direction: the word aggregators LimInf and LimSup are at least as expressive as Inf and Sup for any run aggregator and any language aggregator. The idea is to duplicate the transition system so that the automaton can switch between the two copies at any step, ensuring that every word yields infinitely many runs with the same value.

▶ **Lemma 7.8.** *Let $\mathbb{A} = (h, (g, f, \mathcal{T}))$ be a QLA with f any run aggregator function, $g = \mathsf{Sup}$ (resp. $g = \mathsf{Inf}$), and h any language aggregator function. We can construct in PTime a QLA $\mathbb{B} = (h, (g', f, \mathcal{T}')$ where $g' = \mathsf{LimSup}$ (resp. $g' = \mathsf{LimInf}$) such that $\mathbb{A}(S) = \mathbb{B}(S)$ for all $S \subseteq \Sigma^\omega$.*

The following comes as a consequence of Lemmas 7.7 and 7.8.

▶ **Theorem 7.9.** *For QLAs with run aggregators $f \in \{\mathsf{Inf}, \mathsf{Sup}, \mathsf{LimInf}, \mathsf{LimSup}\}$, the word aggregators* Inf *and* LimInf *(resp.* Sup *and* LimSup*) are equally expressive.*

Finally, we turn our attention to language aggregators. In contrast to the case of word aggregators, even when combined with run aggregators $f \in \{\mathsf{Inf}, \mathsf{Sup}, \mathsf{LimInf}, \mathsf{LimSup}\}$, the language aggregators Inf and Sup are expressively incomparable with LimInf and LimSup. Intuitively, when the top or the bottom value of the underlying QWA is achievable by a single word, a QLA with a limit language aggregator cannot achieve this value, while one with a non-limit language aggregator can. Conversely, if the extreme value of a QLA emerges only as the value of finite languages, a limit language aggregator can capture this behavior, while a non-limit aggregator cannot.

▶ **Proposition 7.10.** *QLAs with the language aggregators* Inf *and* LimInf *(resp.* Sup *and* LimSup*) are expressively incomparable.*

7.3 Decision Problems with Limit Aggregators

Finally, we consider the evaluation, nonemptiness, and universality problems for these classes of QLAs. We first provide a PSpace procedure for evaluation. Intuitively, each weight appearing on the underlying transition system are treated individually since the value of an input language must be one of them. For $h \in \{\mathsf{LimSup}, \mathsf{LimInf}\}$, we leverage Corollary 7.2 to identify the weights achievable by infinitely many words from the input languages in PSpace. The problem reduces to checking the nonemptiness of a Büchi automaton when $h \in \{\mathsf{LimSup}, \mathsf{Sup}\}$ and to checking their inclusion when $h \in \{\mathsf{LimInf}, \mathsf{Inf}\}$. We note that the proof of Theorem 7.11 also establishes the PTime cases presented in Theorem 5.1.

▶ **Theorem 7.11.** *Consider a QLA $\mathbb{A} = (h, (g, f, \mathcal{T}))$ with $f, g, h \in \{\mathsf{Inf}, \mathsf{Sup}, \mathsf{LimInf}, \mathsf{LimSup}\}$ and at least one g and h belong to $\{\mathsf{LimInf}, \mathsf{LimSup}\}$. Let $S \subseteq \Sigma^\omega$ be an ω-regular language given by a Büchi automaton. The value $\mathbb{A}(S)$ is computable in PSpace.*

Next, we prove that QLAs with $f, g, h \in \{\mathsf{Inf}, \mathsf{Sup}, \mathsf{LimInf}, \mathsf{LimSup}\}$, checking whether a given QLA is an upper bound on another (namely, their inclusion problem) is decidable in PSpace. The proof starts by showing that this problem can be solved while reasoning exclusively on ω-regular languages and then generalizes our algorithm for the evaluation problem (Theorem 5.1) to handle two QLAs.

▶ **Theorem 7.12.** *Consider two QLAs $\mathbb{A} = (h, (g, f, \mathcal{T}))$ and $\mathbb{B} = (h', (g', f', \mathcal{T}'))$ with $f, f', g, g', h, h' \in \{\mathsf{Inf}, \mathsf{Sup}, \mathsf{LimInf}, \mathsf{LimSup}\}$. Let $\triangleright \in \{>, \geq\}$. Deciding whether $\mathbb{A}(S) \triangleright \mathbb{B}(S)$ for every language $S \subseteq \Sigma^\omega$ is in PSpace. The same holds when S ranges over ω-regular languages.*

Using Theorem 7.12, we obtain a matching solution to the corresponding nonemptiness and universality problems. Note that Theorems 6.2 and 6.3 capture the cases of non-limit aggregators, including the PTime fragments.

▶ **Corollary 7.13.** *Consider a QLA* $\mathbb{A} = (h, (g, f, \mathcal{T}))$ *with* $f, g, h \in \{\mathsf{Inf}, \mathsf{Sup}, \mathsf{LimInf}, \mathsf{LimSup}\}$ *and at least one* g *and* h *belong to* $\{\mathsf{LimInf}, \mathsf{LimSup}\}$*. Let* $\triangleright \in \{>, \geq\}$*. The* $\triangleright$*-nonemptiness (resp.* $\triangleright$*-universality) of* $\mathbb{A}$ *is in PSpace. The statement holds also for the finite-state restriction.*

8 Conclusion

We introduced quantitative language automata (QLAs) as a uniform framework for specifying and verifying quantitative hyperproperties. Our framework extends beyond both the traditional boolean perspective of system properties and the "one-trace limitation" of traditional quantitative properties as system specifications, enabling reasoning about quantitative aspects of system-wide behaviors such as performance and robustness. We established a thorough foundation for QLAs by investigating the evaluation, nonemptiness, and universality problems, for which we provided a extensive picture of decidability results. In addition to closing the finite-state cases we left open, future research directions include exploring aggregators capable of specifying richer relational system properties, investigating decidable expressive fragments of QLAs, studying equivalent logical formalisms, and augmenting the software tool Quantitative Automata Kit (QuAK) [7,8] with a support for QLAs.

References

1 Shaull Almagor, Udi Boker, and Orna Kupferman. What's decidable about weighted automata? *Inf. Comput.*, 282:104651, 2022. doi:10.1016/J.IC.2020.104651.

2 Bowen Alpern and Fred B. Schneider. Defining liveness. *Inf. Process. Lett.*, 21(4):181–185, 1985. doi:10.1016/0020-0190(85)90056-0.

3 Daniel Andersson. An improved algorithm for discounted payoff games. In Janneke Huitink and Sophia Katrenko, editors, *Proceedings of the 11th ESSLLI Student Session*, pages 91–98, August 2006.

4 Udi Boker, Thomas A. Henzinger, Nicolas Mazzocchi, and N. Ege Saraç. Safety and liveness of quantitative automata. In Guillermo A. Pérez and Jean-François Raskin, editors, *34th International Conference on Concurrency Theory, CONCUR 2023, September 18-23, 2023, Antwerp, Belgium*, volume 279 of *LIPIcs*, pages 17:1–17:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICS.CONCUR.2023.17.

5 Udi Boker, Thomas A Henzinger, Nicolas Mazzocchi, and N Ege Saraç. Safety and liveness of quantitative properties and automata. *Logical Methods in Computer Science*, 21, 2025.

6 Udi Boker, Thomas A. Henzinger, and Jan Otop. The target discounted-sum problem. In *30th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2015, Kyoto, Japan, July 6-10, 2015*, pages 750–761. IEEE Computer Society, 2015. doi:10.1109/LICS.2015.74.

7 Marek Chalupa, Thomas A. Henzinger, Nicolas Mazzocchi, and N. Ege Saraç. Quak: Quantitative automata kit. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Software Engineering Methodologies - 12th International Symposium, ISoLA 2024, Crete, Greece, October 27-31, 2024, Proceedings, Part IV*, volume 15222 of *Lecture Notes in Computer Science*, pages 3–20. Springer, 2024. doi:10.1007/978-3-031-75387-9_1.

8 Marek Chalupa, Thomas A. Henzinger, Nicolas Mazzocchi, and N. Ege Saraç. Automating the analysis of quantitative automata with quak. In Arie Gurfinkel and Marijn Heule, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 31st International*

Conference, TACAS 2025, Held as Part of the International Joint Conferences on Theory and Practice of Software, ETAPS 2025, Hamilton, ON, Canada, May 3-8, 2025, Proceedings, Part I, volume 15696 of *Lecture Notes in Computer Science*, pages 303–312. Springer, 2025. doi:10.1007/978-3-031-90643-5_16.

9 Krishnendu Chatterjee, Laurent Doyen, Herbert Edelsbrunner, Thomas A. Henzinger, and Philippe Rannou. Mean-payoff automaton expressions. In Paul Gastin and François Laroussinie, editors, *CONCUR 2010 - Concurrency Theory, 21th International Conference, CONCUR 2010, Paris, France, August 31-September 3, 2010. Proceedings*, volume 6269 of *Lecture Notes in Computer Science*, pages 269–283. Springer, 2010. doi:10.1007/978-3-642-15375-4_19.

10 Krishnendu Chatterjee, Laurent Doyen, and Thomas A. Henzinger. Alternating weighted automata. In Miroslaw Kutylowski, Witold Charatonik, and Maciej Gebala, editors, *Fundamentals of Computation Theory, 17th International Symposium, FCT 2009, Wroclaw, Poland, September 2-4, 2009. Proceedings*, volume 5699 of *Lecture Notes in Computer Science*, pages 3–13. Springer, 2009. doi:10.1007/978-3-642-03409-1_2.

11 Krishnendu Chatterjee, Laurent Doyen, and Thomas A. Henzinger. Expressiveness and closure properties for quantitative languages. *Log. Methods Comput. Sci.*, 6(3), 2010. URL: http://arxiv.org/abs/1007.4018.

12 Krishnendu Chatterjee, Laurent Doyen, and Thomas A. Henzinger. Quantitative languages. *ACM Trans. Comput. Log.*, 11(4):23:1–23:38, 2010. doi:10.1145/1805950.1805953.

13 Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *J. Comput. Secur.*, 18(6):1157–1210, 2010. doi:10.3233/JCS-2009-0393.

14 Aldric Degorre, Laurent Doyen, Raffaella Gentilini, Jean-François Raskin, and Szymon Torunczyk. Energy and mean-payoff games with imperfect information. In Anuj Dawar and Helmut Veith, editors, *Computer Science Logic, 24th International Workshop, CSL 2010, 19th Annual Conference of the EACSL, Brno, Czech Republic, August 23-27, 2010. Proceedings*, volume 6247 of *Lecture Notes in Computer Science*, pages 260–274. Springer, 2010. doi:10.1007/978-3-642-15205-4_22.

15 Emmanuel Filiot, Nicolas Mazzocchi, and Jean-François Raskin. A pattern logic for automata with outputs. In Mizuho Hoshi and Shinnosuke Seki, editors, *Developments in Language Theory - 22nd International Conference, DLT 2018, Tokyo, Japan, September 10-14, 2018, Proceedings*, volume 11088 of *Lecture Notes in Computer Science*, pages 304–317. Springer, 2018. doi:10.1007/978-3-319-98654-8_25.

16 Bernd Finkbeiner, Christopher Hahn, and Hazem Torfah. Model checking quantitative hyperproperties. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*, volume 10981 of *Lecture Notes in Computer Science*, pages 144–163. Springer, 2018. doi:10.1007/978-3-319-96145-3_8.

17 Vojtech Havlena, Ondrej Lengál, and Barbora Smahlíková. Complementing Büchi automata with ranker. In Sharon Shoham and Yakir Vizel, editors, *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part II*, volume 13372 of *Lecture Notes in Computer Science*, pages 188–201. Springer, 2022. doi:10.1007/978-3-031-13188-2_10.

18 Thomas A. Henzinger, Nicolas Mazzocchi, and N. Ege Saraç. Quantitative safety and liveness. In Orna Kupferman and Pawel Sobocinski, editors, *Foundations of Software Science and Computation Structures - 26th International Conference, FoSSaCS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22-27, 2023, Proceedings*, volume 13992 of *Lecture Notes in Computer Science*, pages 349–370. Springer, 2023. doi:10.1007/978-3-031-30829-1_17.

19 Thomas A. Henzinger and Jan Otop. From model checking to model measuring. In Pedro R. D'Argenio and Hernán C. Melgratti, editors, *CONCUR 2013 - Concurrency Theory - 24th International Conference, CONCUR 2013, Buenos Aires, Argentina, August 27-30, 2013.*

Proceedings, volume 8052 of *Lecture Notes in Computer Science*, pages 273–287. Springer, 2013. doi:10.1007/978-3-642-40184-8_20.

20 Paul Hunter, Arno Pauly, Guillermo A. Pérez, and Jean-François Raskin. Mean-payoff games with partial observation. *Theor. Comput. Sci.*, 735:82–110, 2018. doi:10.1016/J.TCS.2017.03.038.

21 Narendra Karmarkar. A new polynomial-time algorithm for linear programming. *Comb.*, 4(4):373–396, 1984. doi:10.1007/BF02579150.

22 Richard M. Karp. A characterization of the minimum cycle mean in a digraph. *Discret. Math.*, 23(3):309–311, 1978. doi:10.1016/0012-365X(78)90011-0.

23 Daniel Krob. The equality problem for rational series with multiplicities in the tropical semiring is undecidable. *Int. J. Algebra Comput.*, 4(3):405–426, 1994. doi:10.1142/S0218196794000063.

24 Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.*, 3(2):125–143, 1977. doi:10.1109/TSE.1977.229904.

25 Christof Löding and Anton Pirogov. On finitely ambiguous Büchi automata. In Mizuho Hoshi and Shinnosuke Seki, editors, *Developments in Language Theory - 22nd International Conference, DLT 2018, Tokyo, Japan, September 10-14, 2018, Proceedings*, volume 11088 of *Lecture Notes in Computer Science*, pages 503–515. Springer, 2018. doi:10.1007/978-3-319-98654-8_41.

26 Omid Madani, Steve Hanks, and Anne Condon. On the undecidability of probabilistic planning and related stochastic optimization problems. *Artif. Intell.*, 147(1-2):5–34, 2003. doi:10.1016/S0004-3702(02)00378-8.

27 Jakub Michaliszyn and Jan Otop. Non-deterministic weighted automata evaluated over Markov chains. *J. Comput. Syst. Sci.*, 108:118–136, 2020. doi:10.1016/J.JCSS.2019.10.001.

28 Shubham Sahai, Pramod Subramanyan, and Rohit Sinha. Verification of quantitative hyperproperties using trace enumeration relations. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I*, volume 12224 of *Lecture Notes in Computer Science*, pages 201–224. Springer, 2020. doi:10.1007/978-3-030-53288-8_11.

29 Sven Schewe. Büchi complementation made tight. In Susanne Albers and Jean-Yves Marion, editors, *26th International Symposium on Theoretical Aspects of Computer Science, STACS 2009, February 26-28, 2009, Freiburg, Germany, Proceedings*, volume 3 of *LIPIcs*, pages 661–672. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Germany, 2009. doi:10.4230/LIPICS.STACS.2009.1854.

30 Andreas Weber and Helmut Seidl. On the degree of ambiguity of finite automata. *Theor. Comput. Sci.*, 88(2):325–349, 1991. doi:10.1016/0304-3975(91)90381-B.

31 Linpeng Zhang, Noam Zilberstein, Benjamin Lucien Kaminski, and Alexandra Silva. Quantitative weakest hyper pre: Unifying correctness and incorrectness hyperproperties via predicate transformers. *CoRR*, abs/2404.05097, 2024. doi:10.48550/arXiv.2404.05097.

A Proofs of Section 7

▶ **Lemma 7.1.** Let $\mathcal{T}$ be a (weighted) labeled transition system with n states and q_I its initial state. For every state q_2 of $\mathcal{T}$, there exist infinitely many words with a run from q_I visiting q_2 infinitely often iff $\mathcal{T}$ complies with the pattern parameterized by q_2 and n given in Figure 2a.

Proof. We start with the backward implication. Given the state q_2, we assume that $\mathcal{T}$ complies with the pattern in Figure 2a exhibiting four finite runs of the form $\rho_1 : q_I \xrightarrow{u_1} q_1$, $\ell_1 : q_1 \xrightarrow{u_2u_3} q_1$, $\rho_2 : q_1 \xrightarrow{v} q_2$, $\ell_2 \colon q_2 \xrightarrow{u_2u_3} q_2$. Additionally we have that $|u_2| = |v|$ and $u_2 \neq v$. For all $i \in \mathbb{N}$, we define the run $\pi_i = \rho_1(\ell_1)^i\rho_2(\ell_2)^\omega$ from q_I over $u_1(u_2u_3)^iv(u_2u_3)^\omega$.

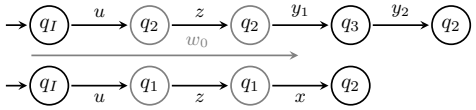


Figure 3 Run synchronization in proof of Lemma 7.1, where $u, x, y_1, y_2, z \in \Sigma^+$ are such that $|x| = |y_1|$, $x \neq y_1$, $|z| = |y_1 y_2|$, and $m|x| = |z|$ for some $m \in \{1, \dots, 4n^4\}$.

Since $|u_2| = |v|$ and $u_2 \neq v$, we have that π_i and π_j read distinct words for all $i \neq j$. In particular, the set $\{u_1(u_2u_3)^i v(u_2u_3)^\omega \mid i \in \mathbb{N}\}$ of ultimately periodic words with a run visiting q_2 infinitely often is infinite.

Next, we prove the forward implication having an infinite set $S \subseteq \Sigma^\omega$ of words admitting some run from the initial state q_I that visits the state q_2 infinitely often. Consider a word $w_1 \in S$ together with one of its runs π_1 that visits q_2 infinitely. Let $w_0 \in \Sigma^*$ be a finite prefix of w_1 for which π_1 visits at least n^2 times the state q_2 while reading w_0, where n refers to the number of states of $\mathcal{T}$. Assuming that w_1 mismatches with all the other words of S exclusively over its $|w_0|$ first letters contradicts $|S| = \infty$. Hence, there exists another word $w_2 \in S$, distinct from w_1, but starting with the finite prefix w_0. Let π_2 be a run over w_2 that visits q_2 infinitely often. Since π_1 visits n^2 times the state q_2 while reading w_0, by the pigeon hole principle, there exists a state q_1 such that the pair (q_1, q_2) is visited twice by (π_1, π_2) while reading w_0. Figure 3 displays the two runs π_1 and π_2. The finite word z refers the factor of w_0 corresponding to the synchronized loop, and the finite word u stands for the prefix of w_0 before z. We ensure that $z \leq n^2$ by removing inner synchronized loops. We denote by x and y_1 the synchronized section where π_1 and π_2 mismatch. In particular, $|x| = |y_1|$ and $x \neq y_1$. Since both π_1 and π_2 visit q_2 infinitely often, we can take x and y_1 long enough so that x leads to q_2. Then, we denote by $\hat{y}_2$ the continuation of y_1 ending in q_2. Additionally, we ensure that $|y_1\hat{y}_2| \leq 3n^2 + 1$. Indeed, any synchronized loop in the factors x and y_1 appearing before or after the mismatching letter can be removed without changing the reachability of q_2. Moreover, any loop in the factor $\hat{y}_2$ can be removed as well. Note that z, x, y_1 are nonempty by construction. By unfolding the synchronized loop, we can assume without loss of generality that all u and $\hat{y}_2$ are nonempty. Hence, all words appearing in Figure 3 are nonempty. From $\hat{y}_2$, we define y_2 as $\hat{y}_2(y_1\hat{y}_2)^{|x|-1}$. This implies that $|y_1y_2| = m|x|$ for some $m \in \{1, \dots, 3n^2 + 1\}$. Furthermore, we can assume that $|z| = |y_1y_2|$, otherwise we lengthen y_2 and z by pumping the synchronized loop over z. In particular $|z| = |y_1y_2| = m|x|$ for some $m \in \{1, \dots, 4n^4\}$.

Now we distinguish the following two cases. First, if $z \neq y_1y_2$, then we define $u_1 = u$, $u_2 = z$, $u_3 = \varepsilon$, and $v = y_1y_2$. Second, if $z = y_1y_2$, then we define $u_1 = u$, $u_2 = y_1$, $u_3 = y_2$, and $v = x$. Either way, u_1, u_2, u_3, v allow $\mathcal{T}$ to comply with the pattern in Figure 2a. ◀

▶ **Lemma A.1.** *Consider a QWA $\mathcal{A} = (g, f, \mathcal{T})$ with n states, $f \in \{\mathsf{LimInf}, \mathsf{LimSup}\}$, $g \in \{\mathsf{Inf}, \mathsf{Sup}, \mathsf{LimInf}, \mathsf{LimSup}\}$. Given $x \in \mathbb{Q}$, we can construct in polynomial time a Büchi automaton $\mathcal{B}$ such that (i) $L(\mathcal{B}) = \{w \in \Sigma^\omega \mid \mathcal{A}(w) = x\}$ and (ii) for every word w, the automaton $\mathcal{B}$ admits infinitely many accepting runs over w iff $\mathcal{A}$ admits infinitely many runs of value x over w.*

Proof. We start by constructing the Büchi automaton $\mathcal{B}_1$ that captures the runs visiting infinitely a transition of weight x. To do so, we consider two copies of $\mathcal{T}$ (the transition system of $\mathcal{A}$) such that: (1.i) the states of the first copy are all accepting, (1.ii) firing any transition from the first copy leads to the second copy, (2.i) the states of the second copy are

all non-initial and non-accepting, (2.ii) firing a transition from the second copy leads to the first copy if and only if its weight is x. Essentially, $\mathcal{B}_1$ visits an accepting state only when a transition weighted x is fired.

Then, from $\mathcal{B}_1$, we construct the Büchi automaton $\mathcal{B}_2$ that captures the runs of value exactly x. To do so, we consider three copies of $\mathcal{B}_1$ such that: (1.i) the states of the first copy are all non-accepting, (1.ii) firing any transition from the first copy leads nondeterministically to the second or the third copy, (2.i) the states of the second copy are all non-initial and non-accepting, (2.ii) firing a transition from the second copy leads to the first copy if and only if its weight is respectively larger or lower than x when $f = \mathsf{LimSup}$ or $f = \mathsf{LimInf}$ (otherwise it stays in the second copy), (3.i) the states of the third copy are all accepting and non-initial, (3.ii) all transitions weighted more than x are removed from the third copy, and the others are unchanged. Essentially, $\mathcal{B}_2$ reaches the copy with accepting states (the third one) only once no more transitions dismissing the value x can be fired.

Observe that there is a bijection between the accepting runs of $\mathcal{B}_2$ and the runs of value x in $\mathcal{A}$. It follows from the fact that $\mathcal{B}_1$ jumps between the copies of $\mathcal{A}$ deterministically and the nondeterministic choices of $\mathcal{B}_2$ in its first copy are unambiguous (in the following sense for $f = \mathsf{LimSup}$: jumping to the second copy and not seeing a weight larger than x in $\mathcal{A}$ and jumping to the third copy and seeing a weight larger than x both result in a rejecting run). Note that this does not hold for rejecting runs due to the nondeterminism of $\mathcal{B}_2$. ◀

▶ **Lemma 7.3.** Consider a QWA $\mathcal{A} = (g, f, \mathcal{T})$ with n states, $f \in \{\mathsf{LimInf}, \mathsf{LimSup}\}$, $g \in \{\mathsf{Inf}, \mathsf{Sup}, \mathsf{LimInf}, \mathsf{LimSup}\}$, and initial state q_I. Let $\sim = \leq$ if $f = \mathsf{LimSup}$ and $\sim = \geq$ if $f = \mathsf{LimInf}$. For every $x \in \mathbb{Q}$ and every lasso word $w \in \Sigma^\omega$, we have that w admits infinitely many runs of value x in $\mathcal{A}$ iff $\mathcal{A}$ complies with the pattern in Figure 2b parameterized by x, n, $\sim$, and w.

Proof. We first prove the backward implication. Given a weight x and a lasso word $w \in \Sigma^\omega$, we assume that $\mathcal{A}$ complies with the pattern given in Figure 2b exhibiting four runs of the form $\rho_1 : q_I \xrightarrow{u_1} q_1$, $\ell_1 : q_1 \xrightarrow{v_1 a v_2} q_1$, $\rho_2 : q_1 \xrightarrow{u_2} q_2$ and $\ell_2 : q_2 \xrightarrow{v_1 a v_2} q_2$ where $q_1 \neq q_2$, $w = u(v_1 a v_2)^\omega$, and $(u_2)^m = v_1 a v_2$ for some $m \in \{1, \ldots, 4n^4\}$. For all $i \in \mathbb{N}$, we define the run $\pi_i = \rho_1(\ell_1)^i \rho_2(\ell_2)^\omega$ in $\mathcal{A}$. Since $(u_2)^m = v_1 a v_2$, the run π_i is over w and achieves the value x thanks to the definition of $\sim$. As a direct consequence of $q_1 \neq q_2$, we have that $\pi_i \neq \pi_j$ for all $i \neq j$. In particular, w have infinitely many runs of value x in $\mathcal{A}$.

Next, we prove the forward implication. Given a weight x and a lasso word $w \in \Sigma^\omega$, we assume that w has infinitely many runs of value x in $\mathcal{A}$. Let $\tilde{u} \in \Sigma^*$ and $\tilde{v} \in \Sigma^* \setminus \{\varepsilon\}$ be such that $w = \tilde{u}(\tilde{v})^\omega$.

In the first step, we construct a Büchi automaton $\mathcal{B}$ that captures the runs over w of value x in $\mathcal{A}$. Thanks to Lemma A.1, we construct the Büchi automaton $\mathcal{B}'$ such that (i) $L(\mathcal{B}') = \{w \in \Sigma^\omega \mid \mathcal{A}(w) = x\}$ and (ii) for every word w, the automaton $\mathcal{B}'$ admits infinitely many accepting runs over w iff $\mathcal{A}$ admits infinitely many runs of value x over w. Finally, from $\mathcal{B}'$, we construction the Büchi automaton $\mathcal{B}$ that captures the runs over w of value x in $\mathcal{A}$. To do so, we consider a deterministic Büchi automaton $\mathcal{B}''$ accepting only w and we construct $\mathcal{B}$ such that $L(\mathcal{B}) = L(\mathcal{B}') \cap L(\mathcal{B}'')$. Observe that there is a bijection between the accepting runs of $\mathcal{B}$ and the runs over w of value x in $\mathcal{A}$. This does not hold for non-accepting runs due to the nondeterminism of $\mathcal{B}'$.

The second step consists of lifting runs to words and leveraging Lemma 7.1 to get the desired pattern. We thus construct a Büchi automaton $\mathcal{C}$ from $\mathcal{B}$ by re-labeling each transition $p_1 \xrightarrow{\sigma} p_2$ by $p_1 \xrightarrow{(p_1,\sigma,p_2)} p_2$. Since w has infinitely many runs in $\mathcal{A}$ and $w \in L(\mathcal{B})$, it has infinitely many accepting runs in $\mathcal{C}$. By Lemma 7.1, some accepting state of $\mathcal{C}$ allows $\mathcal{A}$ to comply with the pattern given in Figure 2a. Consequently there exist $u_1, u_2, u_3, v \in \Sigma^+$ such

that $|u_2| = |v|$, $u_2 \neq v$, and $m|v| = |u_2u_3|$ for some $m \in \{1, \dots, 4n^4\}$, together with four runs of the form $\rho_1 : q_I \xrightarrow{u_1} q_1$, $\ell_1 : q_1 \xrightarrow{u_2u_3} q_1$, $\rho_2 = q_1 \xrightarrow{v} q_2$, $\ell_2 : q_2 \xrightarrow{u_2u_3} q_2$ in $\mathcal{C}$. Observe that u_1, u_2, u_3, v are not over Σ^* but over the alphabet consisting of the transitions of $\mathcal{B}$, called Γ here after. For all finite words $w_0 \in \Gamma^*$ we denote $\Sigma(w_0) \in \Sigma^*$ the projection of all letters of Γ to Σ, e.g., $\Sigma((p_1, \sigma_1, p_2)(p_2, \sigma_2, p_3)) = \sigma_1\sigma_2$. We established that $\mathcal{C}$ admits infinitely many ultimately periodic words of Γ^ω visiting q_2 infinitely often, and thus $\mathcal{B}$ admits infinitely many ultimately periodic runs (over its singleton language $\{w\}$). In particular, the set $\{u_1(u_2u_3)^i v(u_2u_3)^\omega \mid i \in \mathbb{N}\}$ of ultimately periodic words with a run visiting q_2 infinitely often is infinite. However, $\{\Sigma(u_1(u_2u_3)^i v)(\Sigma(u_2u_3))^\omega \mid i \in \mathbb{N}\} = \{w\}$. It is worth emphasizing that despite $u_2 \neq v$ we have $\Sigma(u_2) = \Sigma(v)$ because $L(\mathcal{B}) = \{w\}$.

In the third step, we prove how to ensure $q_1 \neq q_2$. If in the above construction we obtain $q_1 = q_2$, called q here after, then we can construct $u_1', u_2', u_2'', u_3', u_3'', v'$ to get the desired property as follows. We identified $\rho_1 : q_I \xrightarrow{u_1} q$, $\rho_2 : q \xrightarrow{v} q$ and $\ell : q \xrightarrow{u_2u_3} q$ in $\mathcal{C}$. We recall that Lemma 7.1 ensures $|u_2| = |v|$ and $u_2 \neq v$. Since $\Sigma(u_2) = \Sigma(v)$, the mismatch is caused by the states of some letter of Γ. Thus, there are two distinct states $q_1' \neq q_2'$ of $\mathcal{C}$ such that $q \xrightarrow{y_1} q_1'$, $q_1' \xrightarrow{y_2} q$, $q \xrightarrow{z_1} q_2'$, $q_2' \xrightarrow{z_2} q$ in $\mathcal{C}$, where $v = y_1y_2$, $u_2 = z_1z_2$, $|y_1| = |z_1|$, and $|y_2| = |z_2|$. Let $u_1' = u_1y_1$, $u_2' = y_2u_2u_3$, $u_2'' = z_2u_3v$, $u_3' = y_1$, $u_3'' = z_1$, and $v' = y_2vz_1$ together with the four runs of the form $\rho_1' : q_I \xrightarrow{u_1'} q_1'$, $\ell_1' : q_1' \xrightarrow{u_2'u_3'} q_1'$, $\rho_2' : q_1' \xrightarrow{v'} q_2'$, $\ell_2' : q_2' \xrightarrow{u_2''u_3''} q_2'$. For all $i \in \mathbb{N}$, we define the run $\pi_i = \rho_1'(\ell_1')^i\rho_2'(\ell_2')^\omega$ over w in $\mathcal{C}$. Observe that $\pi_i = \rho_1(\rho_2\ell)^i\rho_2\rho_2(\ell\rho_2)^\omega$ because $q_1 = q_2 = q$. In particular, the set $\{u_1'(u_2'u_3')^i v'(u_2''u_3'')^\omega \mid i \in \mathbb{N}\}$ of ultimately periodic words with a run visiting q_2' infinitely often is infinite. We now prove that new pattern complies with Figure 2b. Since $\Sigma(u_2) = \Sigma(v)$, $|y_1| = |z_1|$, and $|y_2| = |z_2|$, we have that $\Sigma(y_1) = \Sigma(z_1)$ and $\Sigma(y_2) = \Sigma(z_2)$. Hence, $\Sigma(u_2'u_3') = \Sigma(u_2''u_3'')$. Additionally, $\Sigma(v)^m = \Sigma(u_2'u_3')$ for some $m \in \{1, \dots, 4n^4\}$ comes as a consequence of $m|v| = |u_2u_3|$ for some $m \in \{1, \dots, 4n^4\}$.

Finally, we decompose u_2u_3 into v_1av_2. We identified $\rho_1 : q_I \xrightarrow{u_1} q_1$, $\ell_1 : q_1 \xrightarrow{u_2u_3} q_1$, $\rho_2 : q_1 \xrightarrow{v} q_2$, and $\ell_2 : q_2 \xrightarrow{u_2u_3} q_2$ where $q_1 \neq q_2$ in $\mathcal{B}$. Recall that there is a bijection between the accepting runs of $\mathcal{B}$ and the runs over w of value x in $\mathcal{A}$. By construction of $\mathcal{B}$, the accepting run $u_1v(u_2u_3)^\omega$ of $\mathcal{B}$ (which is a word accepted by $\mathcal{C}$) corresponds to a run over w of value x in $\mathcal{A}$. Therefore, the loop $\ell_2 : q_2 \xrightarrow{u_2u_3} q_2$ corresponds to a loop visiting the weight x and other weights of value at most (resp. at least) x when $f = \mathsf{LimSup}$ (resp. $f = \mathsf{LimInf}$). To conclude, there exist $v_1, v_2 \in \Sigma^*$ and $a \in \Sigma$ such that $u_2u_3 = v_1av_2$ together with the runs $q_2 \xrightarrow{v_1 : \sim x} q_3$, $q_4 \xrightarrow{v_2 : \sim x} q_2$ and the transition $q_3 \xrightarrow{a : x} q_4$. ◀

▶ **Lemma 7.7.** Let $\mathbb{A} = (h, (g, f, \mathcal{T}))$ be a QLA with $f \in \{\mathsf{Inf}, \mathsf{Sup}, \mathsf{LimInf}, \mathsf{LimSup}\}$, $g = \mathsf{LimSup}$ (resp. $g = \mathsf{LimInf}$), and h any language aggregator function. We can construct in PSPACE a QLA $\mathbb{B} = (h, (g', f', \mathcal{T}'))$ with $f' = \mathsf{LimSup}$ and $g' = \mathsf{Sup}$ (resp. $f' = \mathsf{LimInf}$ and $g' = \mathsf{Inf}$) such that $\mathbb{A}(S) = \mathbb{B}(S)$ for all $S \subseteq \Sigma^\omega$.

Proof. Consider a language automaton $\mathbb{A} = (h, \mathcal{A})$ where $\mathcal{A} = (g, f, \mathcal{T})$ is as in the statement. We show the case $g = \mathsf{LimSup}$, as the case $g = \mathsf{LimInf}$ can be solved by duality (Proposition 2.2). Thanks to Proposition 7.5, we can assume that $f \in \{\mathsf{LimInf}, \mathsf{LimSup}\}$. The proof describes the construction of a QWA $\mathcal{B} = (\mathsf{Sup}, f, \mathcal{T}')$ such that $\mathcal{A}(w) = \mathcal{B}(w)$ for every ultimately periodic word $w \in \Sigma^\omega$, and argue why this equivalence generalizes to all words.

Recall that, by definition (when $g = \mathsf{LimSup}$), $\bot_{\mathcal{A}}$ is the smallest weight x in $\mathcal{T}$ such that there is a word w with infinitely many runs of value x in $\mathcal{T}$ and at most finitely many runs of value larger than x in $\mathcal{T}$. Therefore, we can compute $\bot_{\mathcal{A}}$ as follows: Take a weight x in $\mathcal{T}$ and construct in PTIME a Büchi automaton $\mathcal{D}_x$ from the transition system $\mathcal{T}$, recognizing exactly the set of words with some run of value x in $\mathcal{T}$. Then, using [15, 25], we can decide in

PTime if $\mathcal{D}_x$ has an infinite degree of ambiguity, i.e., some word in its language has infinitely many accepting runs (namely, runs of value x in $\mathcal{T}$). Let Y be the set of weights x such that $\mathcal{D}_x$ has an infinite degree of ambiguity. Then, $\bot_{\mathcal{A}}$ is the largest $y \in Y$ such that the automaton $\mathcal{D} = (\mathsf{Sup}, f, \mathcal{T})$ is $\geq$-universal for y, which can be computed in PSpace. We can compute $\bot_{\mathcal{A}}$ similarly when $g = \mathsf{LimInf}$. Now, let X be the set of all weights in $\mathcal{T}$. By Corollary 7.4, for all $x \in X$, we can construct in PTime a Büchi (if $f = \mathsf{LimSup}$) or a co-Büchi (if $f = \mathsf{LimInf}$) automaton $\mathcal{C}_x$ recognizing all ultimately periodic words admitting infinitely many runs of value x in $\mathcal{A}$. From $\mathcal{C}_x$ we construct in PTime the QWA $\mathcal{A}'_x = (\mathsf{Sup}, f, \mathcal{T}_x)$ such that $\mathcal{A}'_x(w) = x$ when $w \in L(\mathcal{C}_x)$ and $\mathcal{A}'_x(w) = \bot_{\mathcal{A}'}$ when $w \notin L(\mathcal{C}_x)$. The construction consists in changing the transition weight 1 to x and 0 to $\bot_{\mathcal{A}}$. The QWA $\mathcal{B} = (\mathsf{Sup}, f, \mathcal{T}')$ is defined by $\mathcal{B} = \max_{x \in X} \mathcal{A}'_x$, effectively computable in PTime [11].

For every ultimately periodic word $w \in \Sigma^\omega$, the value $\mathcal{A}(w)$ corresponds to the maximal value $x \in X$ such that there exist infinitely many runs of value x for w in $\mathcal{T}$. Let $X_w \subseteq X$ denote the set of values x for which w admits infinitely many runs in $\mathcal{T}$; equivalently, $x \in X_w$ iff $w \in L(C_x)$. By construction, we have $\mathcal{A}'_x(w) = x$ for each $x \in X_w$ and $\mathcal{A}'_x(w) = \bot_{\mathcal{A}'}$ for each $x \in X \setminus X_w$. If $X_w \neq \emptyset$, it follows that $\mathcal{A}(w) = \max X' = \max_{x \in X'} \mathcal{A}'_x(w) = \mathcal{B}(w)$. Otherwise, if $X_w = \emptyset$, we have $\mathcal{A}(w) = \bot_{\mathcal{A}'} = \mathcal{B}(w)$.

Next, we argue that if $\mathcal{A}(w) = \mathcal{B}(w)$ for every ultimately periodic word $w \in \Sigma^\omega$, then $\mathcal{A}(w) = \mathcal{B}(w)$ for every word $w \in \Sigma^\omega$. We prove the contrapositive. Suppose the existence of a word $\hat{w}$ such that $\mathcal{A}(\hat{w}) \neq \mathcal{B}(\hat{w})$. Clearly, we have $\mathcal{A}(\hat{w}) \in X$ and $\mathcal{B}(\hat{w}) \in X$. Using Lemma A.1, we construct a Büchi automaton $\mathcal{A}'_{\hat{w}}$ such that (i) $L(\mathcal{A}'_{\hat{w}}) = \{w \in \Sigma^\omega \mid \mathcal{A}'(w) = \mathcal{A}'(\hat{w})\}$ and (ii) for every word w, the automaton $\mathcal{A}'_{\hat{w}}$ admits infinitely many accepting runs over w iff $\mathcal{A}'$ admits infinitely many runs of value $\mathcal{A}'(\hat{w})$ over w. Similarly, we construct a Büchi automaton $\mathcal{B}_{\hat{w}}$ such that (i) $L(\mathcal{B}_{\hat{w}}) = \{w \in \Sigma^\omega \mid \mathcal{B}(w) = \mathcal{B}(\hat{w})\}$ and (ii) for every word w, the automaton $\mathcal{B}_{\hat{w}}$ admits infinitely many accepting runs over w iff $\mathcal{B}$ admits infinitely many runs of value $\mathcal{B}(\hat{w})$ over w. Thanks to [25], we can construct from $\mathcal{B}_{\hat{w}}$ an unambiguous Büchi automaton $\mathcal{B}'_{\hat{w}}$ recognizing the same language. The cross product between $\mathcal{A}'_{\hat{w}}$ and $\mathcal{B}'_{\hat{w}}$ yields a Büchi automaton $\mathcal{C}$ such that (i) $L(C) = L(\mathcal{B}_{\hat{w}}) \cap L(\mathcal{A}'_{\hat{w}})$ and (ii) for every word w, the automaton $\mathcal{C}$ admits infinitely many accepting runs over w iff $\mathcal{A}'$ admits infinitely many runs of value $\mathcal{A}'(\hat{w})$ over w. Observe that $\hat{w} \in L(\mathcal{C})$ and for all $w \in L(\mathcal{C})$ we have $\mathcal{A}'(w) = \mathcal{A}'(\hat{w}) \neq \mathcal{B}(\hat{w}) = \mathcal{B}(w)$. By [25, 30], we can decide whether $\mathcal{C}$ admits infinitely many accepting runs on some word. Furthermore, it follows from [25, 30] that if $\mathcal{C}$ admits infinitely many accepting runs on a word, then it recognizes an ultimately periodic word $\tilde{w}$ with infinitely many accepting runs. In this case, $\mathcal{A}'$ admits infinitely many runs of value $\mathcal{A}'(\hat{w})$ over $\tilde{w}$, implying that $\mathcal{A}(\tilde{w}) = \mathcal{A}'(\tilde{w}) = \mathcal{A}'(\hat{w})$. Otherwise, we have $\mathcal{A}(\hat{w}) = \bot_{\mathcal{A}'} = \mathcal{A}'(\hat{w})$, and since $L(\mathcal{C}) \neq \emptyset$, there exists an ultimately periodic word $\tilde{w}$ for which $\mathcal{A}'(\tilde{w}) = \bot_{\mathcal{A}'}$. In either scenario, since $\tilde{w} \in L(\mathcal{C})$, we have $\mathcal{B}(\tilde{w}) = \mathcal{B}(\hat{w})$. Therefore, we conclude that there is an ultimately periodic word $\tilde{w}$ satisfying $\mathcal{A}'(\tilde{w}) \neq \mathcal{B}(\tilde{w})$. ◀

Partial-Order Reduction Is Hard

Frédéric Herbreteau
Univ. Bordeaux, CNRS, Bordeaux INP, LaBRI, UMR 5800, F-33400 Talence, France

Sarah Larroze-Jardiné
Univ. Bordeaux, CNRS, Bordeaux INP, LaBRI, UMR 5800, F-33400 Talence, France

Igor Walukiewicz
Univ. Bordeaux, CNRS, Bordeaux INP, LaBRI, UMR 5800, F-33400 Talence, France

Abstract

The goal of partial-order methods is to accelerate the exploration of concurrent systems by examining only a representative subset of all possible runs. The stateful approach builds a transition system with representative runs, while the stateless method simply enumerates them. The stateless approach may be preferable if the transition system is tree-like; otherwise, the stateful method is more effective.

In the last decade, optimality has been a guiding principle for developing stateless partial-order reduction algorithms, and without doubt contributed to big progress in the field. In this paper we ask if we can get a similar principle for the stateful approach. We show that in stateful exploration, a polynomially close to optimal partial-order algorithm cannot exist unless P=NP. The result holds even for acyclic programs with just await instructions. This lower bound result justifies systematic study of heuristics for stateful partial-order reduction. We propose a notion of IFS oracle as a useful abstraction. The oracle can be used to get a very simple optimal stateless algorithm, which can then be adapted to a non-optimal stateful algorithm. While in general the oracle problem is NP-hard, we show a simple case where it can be solved in linear time.

2012 ACM Subject Classification Theory of computation → Logic and verification

Keywords and phrases Formal verification, Concurrent systems, Partial-order reduction, Complexity

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2025.22

Funding *Frédéric Herbreteau*: Supported by the French government in the framework of the France 2030 programme IdEx université de Bordeaux / RRI ROBSYS.
Igor Walukiewicz: Supported by ANR grant PaVeDys, ANR-23-CE48-0005.

1 Introduction

The goal of partial-order methods is to speed up explicit state exploration of concurrent systems. The state space of such systems grows exponentially with the number of processes. Fortunately, many runs of a concurrent system can usually be considered equivalent, so it is enough to explore only one run in each equivalence class. For example, if one process assigns $x := 2$ and another $y := 3$ then the order of execution of these two operations is usually irrelevant; the two interleavings are equivalent, and it is enough to explore only one of the two. This reduces the number of visited states as well as the exploration time. In some cases, the reductions are very substantial.

In recent years, we have seen novel applications of partial-order methods. One is proving the properties of concurrent programs [10], where equivalence between runs is not only used to reduce the number of proof objectives, but also to simplify proof objectives by choosing particular linearizations. Similarly, in symbolic executions [9] or in testing [28], partial-order can be used to limit the exploration space while still being exhaustive. Another application is verification of timed systems using recently developed local-time zones [15]. All these applications rely on explicit state enumeration, as opposed to symbolic methods such as SAT or BDDs.

© Frédéric Herbreteau, Sarah Larroze-Jardiné, and Igor Walukiewicz;
licensed under Creative Commons License CC-BY 4.0

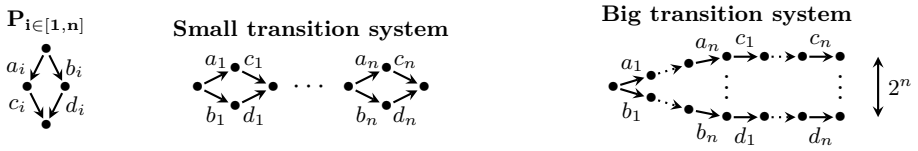


Figure 1 Consider a concurrent system consisting of processes $P_1, \ldots, P_n$, with no dependency between actions. Two trace-optimal transition systems, one of linear size and one of exponential size.

The first partial-order methods were proposed about 35 years ago under the names of stubborn sets, persistent sets, or ample sets [34, 14, 20]. In 2005, Flanagan and Godefroid introduced *stateless* dynamic partial-order reductions [12], making stateless methods a focal point of subsequent research. In 2014, Abdulla et al. proposed a notion of trace-optimality[1] [1], and a race reversal technique, initiating a new cycle of work on stateless partial-order methods [6, 5, 18, 22, 23, 3, 21, 24, 4].

In this paper, we focus on *stateful* partial-order reduction methods, that have seen relatively less progress over the last two decades [29, 37, 8, 7]. A partial-order algorithm produces a *reduced transition system* containing a representative for each equivalence class of runs of the system. A *stateless* approach produces a tree of runs, but saves memory by storing only one run at a time. In contrast, a *stateful* approach keeps all visited states in memory so that an exploration can be stopped if a state is revisited. Each of the two approaches has its advantages. When the objective is to verify existing code by running an instrumentation of it, the states are too complex to keep in memory, so stateless exploration is the only solution. When verifying pseudocode, say of a mutual exclusion protocol, the number of non-equivalent runs may be several orders of magnitude bigger than the number of states, rendering the stateless approach infeasible. The stateful approach is then necessary.

Our first question is whether we can find a guiding principle for the stateful approach similar to the trace-optimality for the stateless approach. The example in Figure 1 shows why we cannot just settle on trace optimality. The two transition systems in the figure are both trace-optimal, since no two maximal paths are equivalent. But one is of linear size and the other is of exponential size. The obvious parameter for optimality of the stateful approach is the size of the transition system. So our question is whether a state-optimal, or close to optimal, partial-order algorithm can exist.

It is not difficult to see that getting a reduced system of exactly the minimal size is NP-hard. Our main result says something much stronger: even approximating the minimal size within a polynomial factor remains NP-hard (Theorem 5). This holds even for acyclic concurrent programs using only await and write instructions. Arguably, this looks like a minimal sensible set of instructions. If we allowed cycles then we could simulate await with active wait, reducing the set of instructions to just reads and writes.

This negative result justifies starting a systematic study of heuristics for stateful partial-order reduction. We propose an approach based on a notion of *IFS oracle*. This oracle expresses an important sub-problem encountered in partial-order reduction algorithms, namely avoiding so called sleep-blocked executions. Our proposal is to concentrate on the heuristics for this oracle to improve the efficiency of stateful methods. We justify this approach by showing that in the stateless setting, the IFS oracle gives a very simple trace-

[1] We use the term trace-optimality instead of simply optimality as in op.cit. to differentiate from state-optimality we consider in this paper.

optimal algorithm. We show how to adapt this algorithm to the stateful approach. Finally, we narrow our study down to a special case, generalizing some settings considered in the stateless literature [5, 22], where the IFS oracle can be computed in linear time.

Before starting, we would like to shortly discuss the models of concurrent systems we are using. In recent years it is common to consider a model of processes with shared variables, eventually adding lock instructions. This is the model we are using for our lower bound. However, we also consider a stronger model of concurrent systems synchronizing on shared actions. This model can encode variables, and most of the synchronization mechanisms used in practice. For this reason we prefer this model for our positive results. Moreover, since the model is strong, our lower bound result is easier to prove for it, so it serves as a good intermediate step in our proof.

In this work we assume that programs are acyclic. Dealing with cycles in partial-order reduction is a separate problem from the reduction itself, and complicates the algorithms substantially. Finding a clean way of dealing with cycles in partial-order reduction is beyond the scope of this paper.

To summarize our three principal contributions are:

- A strong lower bound showing that, assuming $P \neq \text{NP}$, no deterministic algorithm can construct a reduced transition system for a concurrent program $\mathbb{P}$, of polynomial size relative to the *minimal size* of a reduced system for $\mathbb{P}$, while working in time polynomial in the size of its input and its output (Theorem 5). This holds even if every process in the program is acyclic and uses only await and write instructions.
- A notion of *IFS oracle* and a simple trace-optimal stateless algorithm using this oracle. We adapt this algorithm to the stateful approach.
- A special subcase of linear fully non-blocking systems, where the IFS oracle can be computed in linear time. This case covers some situations considered in the stateless partial-order reduction literature.

1.1 Related Work

The literature on partial-order methods has expanded rapidly in the last decade. In this brief discussion, we focus only on results that are closely related to our work, primarily citing more recent papers.

Partial-order methods have been introduced around 1990 [34, 13, 20]. Applications requiring stateful partial-order methods [11] are still based on the same basic principles. Another significant concept for us here is the use of lexicographic ordering to identify representative runs [36, 19, 35]. We also highlight a Petri net unfolding-based approach [32, 8] that distinguishes itself by using prime event structures instead of transition systems.

There are already two NP-hardness results in the literature on partial-order methods. The first is the NP-hardness of computing optimal ample sets [30]. The second is NP-hardness of computing so-called alternatives in the context of unfolding based methods [8]. These results can be compared to our observation (Proposition 22) that IFS problem is NP-complete. Our main negative result is much stronger. It does not address a particular method of doing partial-order reduction, but shows that any method must be NP-hard, even when instead of optimality we just want to be polynomially close to optimal.

Stateless partial-order reduction, initiated by [12], gained momentum thanks to [1]. This approach has been the focus of extensive research in recent years, culminating in truly stateless algorithms using only polynomial-sized memory [22, 3]. Extensions of stateless techniques to systems admitting blocking, like some synchronization mechanisms, have been proposed very recently [18, 23]. Await instructions we use in our lower bound are blocking. We consider non-blocking systems in our positive result in Proposition 24.

The race reversal technique introduced in [1] has been partially adapted for use in stateful methods [37, 7]. Apart from that, the stateful methods still rely on the original stubborn/persistent set approach. The unfolding based partial-order methods have seen developments like the k-partial alternatives [8]. Although IFS oracle comes from standard concepts and problems in partial-order methods, we hope this notion will help to advance the stateful approach.

Apart from typical programs with variables, partial-order methods have been adapted to various frameworks, including actor programs [33], event driven programs [2, 17, 25], and MPI programs [31]. This motivates us to use the shared actions model in our positive results.

2 Modeling concurrent systems

For our lower bounds we will consider a very simple standard model of concurrent programs with variables and locks. Variables range over finite domains, and locks are binary. The only operations on variables are write and await. The later operation blocks until the value of a variable is in a specified set of values. We do not need a read operation in our constructions, so we do not mention it to simplify the presentation.

A *process* is a directed acyclic graph with edges labeled by instructions. The instructions come from the set:

$$\Sigma ::= w(x,i) \mid await(x,I) \mid acq(l) \mid rel(l)$$

Here, $w(x,i)$ sets the value of variable x to i, $await(x,I)$ blocks until the value of x is in the set of values I, $acq(l)$ acquires lock l, and $rel(l)$ releases it. In case of acyclic processes, it is natural to have the blocking *await* instruction. For example, all classical mutual exclusion algorithms rely on it. If we had cycles, *await* could be simulated with an active wait.

A *concurrent program* $\mathbb{P}$ is a finite set of processes, over finite sets of variables and locks. Each variable has a finite domain. All variables and locks are global, meaning that they are shared by all processes. A conditional instruction is modeled by branching in the processes. The semantics of a program $\mathbb{P}$ is given by a transition system TS($\mathbb{P}$). Its states consist of a local state for each process, and a value for each variable and lock. In the initial state, each process is in its initial local state, and each variable and lock have value 0. The transitions in TS($\mathbb{P}$) are defined according to the usual semantics of instructions from Σ. Observe that TS($\mathbb{P}$) is acyclic since each process is acyclic.

We will also consider a much more flexible and expressive model, that we call *concurrent system*. It consists of a collection of processes synchronizing on common actions. Synchronization on actions can directly encode variables, locks, and many other synchronization primitives. In the context of this paper it will be easier to show the lower bound in this model than in the model with variables and locks. But, since the model is strong the lower bound result for this model is relatively weak. This is why we will eventually refine the lower bound construction to concurrent programs without locks.

A concurrent system $\mathbb{S}$ is determined by an alphabet Σ of abstract actions, and a set of processes that are graphs with edges labeled by actions. A process must be *action deterministic*: for every $s \in S$ and $a \in \Sigma$ there is at most one t with $s \xrightarrow{a} t$. The term process is used both in the case of concurrent systems, and concurrent programs, but it will always be clear from the context which model we are using.

Every action a has its *domain*, $dom(a)$, that is a set of processes using it. We write $s_p \xrightarrow{b}_p$ if there is an outgoing b-transition from state s_p of process p. For a sequence of actions v we write $dom(v)$ for $\bigcup\{dom(a) : a \text{ appears in } v\}$, and $s_p \xrightarrow{v}_p$ if there is a path labeled v from s_p in process p.

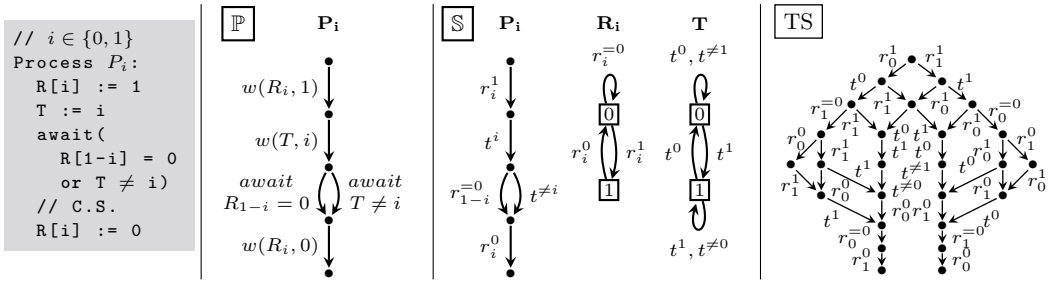


Figure 2 Peterson's mutex algorithm with 2 processes: pseudocode (left), modeled as a concurrent program $\mathbb{P}$ (middle left), as a concurrent system $\mathbb{S}$ (middle right), and their semantics TS (right).

The semantics of a concurrent system $\mathbb{S}$ is a transition system TS($\mathbb{S}$) whose states are tuples of states of process transition systems, $S = \prod_{p \in Proc} S_p$; the initial state is the tuple consisting of initial states of each process, $s^0 = \{s_p^0\}_{p \in Proc}$; every action a synchronizes processes involved in it: $s \xrightarrow{a} s'$ if $s_p \xrightarrow{a}_p s'_p$ for $p \in dom(a)$, and $s'_p = s_p$ for $p \notin dom(a)$. We write $enabled(s)$ for the set of actions labeling transitions outgoing from the global state s of TS($\mathbb{S}$).

Finally, we impose an *acyclicity condition*: for every action a, at least one of the processes in $dom(a)$ must be acyclic. This guarantees that the global transition system is acyclic.

We will need to talk about runs in a transition system TS, be it TS($\mathbb{P}$) for a concurrent program $\mathbb{P}$ or TS($\mathbb{S}$) for a concurrent system $\mathbb{S}$. A *run* is a path in the transition system TS, not necessarily from the initial state. We write $s \xrightarrow{v} t$ if there is a run labeled with a sequence of actions v from s to t. Sometimes we write just $s \xrightarrow{v}$ when t is not relevant. A *maximal run* is a run reaching a terminal state – a state with no outgoing transitions. A *full run* is a maximal run starting in the initial state.

Figure 2 on the left shows Peterson's mutual exclusion algorithm for 2 processes. The middle-left picture shows its modeling as a concurrent program $\mathbb{P} = \{P_0, P_1\}$. It has three variables: R_0, R_1 and T that range over $\{0, 1\}$ as in the pseudocode. The middle-right picture shows the same algorithm modeled as a concurrent system $\mathbb{S} = \{P_0, P_1, R_0, R_1, T\}$. Processes P_0 and P_1 implement the code while the other processes model the variables $R[0]$, $R[1]$ and T. Action r_i^j writes the value j to variable $R[i]$. Action $r_i^{=0}$ checks if the variable $R[i]$ has value 0; observe that it is enabled in the top state where $R[i]$ has value 0, but not in the bottom state where it has value 1. Similarly, for actions of T. Processes in $\mathbb{S}$ synchronize on common actions. For instance, r_0^1 synchronizes P_0 and R_0, while $t^{\neq 1}$ synchronizes P_1 with T. We write this as $dom(r_0^1) = \{P_0, R_0\}$ and $dom(t^{\neq 1}) = \{P_1, T\}$. The transition system TS is shown on the right. It represents both TS($\mathbb{P}$) and TS($\mathbb{S}$), although we have chosen to show labels from $\mathbb{S}$. Each full run corresponds to an execution of Peterson's algorithm.

3 Partial-order reduction

The initial motivation for partial-order reduction comes from the observation that some sequences of actions may be considered equivalent, and only one of the equivalent sequences needs to be explored. The equivalence relation on sequences is defined in terms of the independence relation on actions: two sequences are *por-equivalent* if one can be obtained from the other by permuting adjacent independent actions. We will use $\approx$ to denote a por-equivalence relation on sequences of actions. As we will see our arguments do not rely on a particular independence relation, as long as two actions using different processes and different resources are independent.

For concurrent programs with variables the classical independence relation says that two actions of different processes are independent if either they use different variables or they use the same variable and both of them are awaits. In particular lock operations are dependent.

For concurrent systems, the most common independence relation is the one coming from Mazurkiewicz trace theory [26]. Two actions $a, b \in \Sigma$ are *independent* if they have disjoint domains: aIb if $dom(a) \cap dom(b) = \emptyset$.

It will also be useful to use the dual concept of dependent actions. Two actions are *dependent*, written aDb, if they are not independent. For example, when $dom(a) \cap dom(b) \neq \emptyset$ in case of Mazurkiewicz independence. These notions are extended to sequences: aIv means that a is independent of all actions in v, and aDv that a is dependent on some action from v. We write Da for the set of actions dependent on a.

The goal of partial-order reduction is to construct, for a given concurrent system $\mathbb{S}$ or a concurrent program $\mathbb{P}$, a reduced transition system representing all full runs in TS($\mathbb{S}$) or TS($\mathbb{P}$) (recall that these are maximal runs from the initial state).

▶ **Definition 1.** *We say that* TS_r *is a* reduced transition system *for* TS *if it is:*
- sound*: every full run of* TS_r *is a full run in* TS*, and*
- complete*: for each full run* u *in* TS *there is a full run* v *in* TS_r *such that* $u \approx v$

A general approach for constructing a reduced transition system is to determine for each state s of TS a covering source set [1]: a subset of enabled actions that is sufficient to explore. For example, if every sequence starting from b is equivalent to a sequence starting from c then we may choose to include only b in the source set. The notion of the *first action* modulo an equivalence relation $\approx$ on sequences is central for partial-order reduction.

$$\mathit{first}(u) = \{b : \exists v.\ bv \approx u\}\ .$$

We want a source set in a state to be large enough to contain a first action of at least one representative from every equivalence class of maximal runs from the state. Using the above definition this can be formulated as: a source set in s should intersect every $\mathit{first}(u)$ for u a maximal run from s. We formalize this as follows.

▶ **Definition 2.** *For a state* s *of* TS *we define* $First(s)$ *as the set of first sets of all maximal runs from* s*:*

$$First(s) = \{\mathit{first}(u) : u \text{ is a maximal run from } s\}\ .$$

A set of actions B *is a* covering source set *in* s *if* $B \cap F \neq \emptyset$ *for every* $F \in First(s)$*.*

In particular, if B contains all enabled actions from s then B is a covering source set at s. Intuitively, smaller covering source sets should give smaller reduced transition systems. This is not always true. A bigger but incomparable w.r.t. set inclusion covering source set may give a better reduction.

Observe that the notion of a covering source set depends on the por-equivalence relation $\approx$, as the definition of *first* depends on it.

Suppose we have an assignment of a set of actions $source(s)$ for every state s of TS. We can use it to restrict the transition relation to transitions allowed by source sets: define $s \overset{a}{\Longrightarrow} t$ when $s \overset{a}{\longrightarrow} t$ and $a \in source(s)$. If every $source(s)$ is a covering source set in s then this restricted transition relation is enough.

▶ **Proposition 3.** *Let* TS *be a finite acyclic transition system with a transition relation* $\longrightarrow$*. Suppose* $\Longrightarrow$ *is the restricted transition relation derived from a covering source set assignment. For every state* s *of* TS*, and every maximal run* $s \overset{u}{\longrightarrow}$ *there is a run* $v \approx u$ *with* $s \overset{v}{\Longrightarrow}$*.*

This proposition allows us to construct a reduced transition system by keeping only $\Longrightarrow$ transitions and states reachable from the initial state using these transitions. We must underline though that source sets are not the only mechanism needed in partial-order reduction. Sleep-sets are another important ingredient (c.f. Remark 21).

4 Partial-order reduction is NP-hard

Recall that the goal of partial-order reduction is to construct for a given program $\mathbb{P}$ or a given system $\mathbb{S}$, a reduced system that is sound and complete. Among such systems we would ideally like to construct one with the smallest number of states. We write $minTS(\mathbb{P})$ for the smallest number of states of a sound and complete reduced transition system for $\mathbb{P}$. Similarly, we write $minTS(\mathbb{S})$ for a system $\mathbb{S}$. There may be several non-isomorphic transition systems with a minimal number of states.

Stateless algorithms use the concept of *trace-optimality*. This means that there are no two equivalent runs in the reduced transition system. As we have noted in the introduction, this concept is not very useful for our purposes. Indeed, the example from Figure 1 shows two trace-optimal transition systems, one of linear size and the other of exponential size. So in stateful partial-order reduction we should aim for a reduced transition system of small size. Even more so as there are also examples where the smallest reduced transition system is necessarily not trace-optimal.

In this section, we show that there is no polynomial-time algorithm for constructing a reduced transition system that is polynomially close to optimal. The next definition makes this precise.

▶ **Definition 4.** *We say that Alg is an* excellent POR algorithm *if there are polynomials $q(x)$ and $r(x)$ such that given a concurrent program $\mathbb{P}$, the algorithm constructs a sound and complete transition system for $\mathbb{P}$ of size bounded by $q(minTS(\mathbb{P}))$ in time $r(|\mathbb{P}| + minTS(\mathbb{P}))$.*

We use the same definition for concurrent systems $\mathbb{S}$.

The main result of this section is

▶ **Theorem 5.** *If P≠NP then there is no excellent POR algorithm, even for concurrent programs using only write and await operations.*

We will prove this theorem in three steps. First we will consider concurrent systems. The flexibility of synchronization operations makes the constructions used in the proof much simpler, and the arguments can be supported by figures of a reasonable size. In the next step we adapt the construction to concurrent programs with locks and await. In the last step we eliminate locks. The proof is structured in such a way that the same statements need to be proved in each of the three steps.

4.1 Synchronizations

The proof of Theorem 5 uses an encoding of the 3-SAT problem φ into a concurrent system $\mathbb{S}_\varphi$. If φ is not satisfiable then $\mathbb{S}_\varphi$ has a small reduced transition system. If it is satisfiable then all reduced transition systems for $\mathbb{S}_\varphi$ are big. With such an encoding we show that an excellent POR algorithm would give a polynomial-time algorithm for deciding 3-SAT.

Consider a 3-CNF formula over literals $\{x_1, \dots, x_n\} \cup \{\bar{x}_1, \dots, \bar{x}_n\}$ and with k clauses:

$$\varphi = (\alpha_1^1 \vee \alpha_2^1 \vee \alpha_3^1) \wedge \cdots \wedge (\alpha_1^k \vee \alpha_2^k \vee \alpha_3^k)$$

where each α_j^i is a literal.

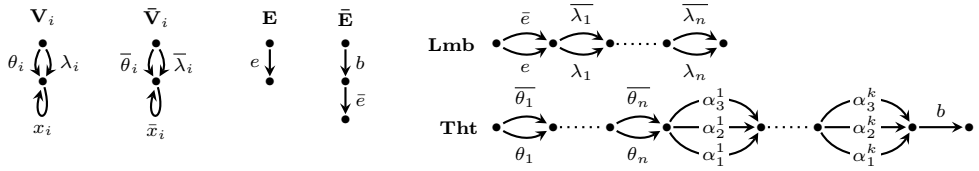


Figure 3 Concurrent system $\mathbb{S}_\varphi$ for the lower bound argument.

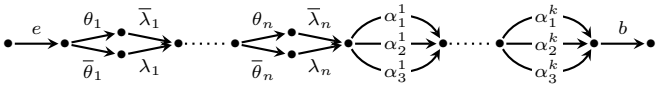


Figure 4 A sound a complete transition system for $\mathbb{S}_\varphi$ when φ is not SAT.

The concurrent system $\mathbb{S}_\varphi$ is presented in Figure 3. It has two processes for each propositional variable, V_i and $\bar{V}_i$, two special processes *Lmb*, and *Tht*, and two auxiliary processes E and $\bar{E}$. For every $i = 1, \ldots, n$ we have a process V_i corresponding to variable x_i. Process V_i has two states: top and bottom. There are two transitions from top to bottom state, the θ_i transition intuitively says that x_i should be true, and λ_i that it should be false. This choice is not encoded in the reached state though, as the two transitions go to the same bottom state where a transition on x_i is possible; interactions with special processes will make the difference. Process $\bar{V}_i$ is similar, but now we have $\bar{\theta}_i$, $\bar{\lambda}_i$, and $\bar{x}_i$ actions.

Apart from the variable processes V_i and $\bar{V}_i$, we have two special processes *Lmb* and *Tht*. Only *Tht* depends on formula φ. Process *Lmb* synchronizes on λ_i actions, and *Tht* on θ_i actions. Process *Lmb* starts with a choice between e and $\bar{e}$, while the second part of *Tht* corresponds to the clauses of the formula φ, and finishes with action b. Finally, there are two auxiliary processes E and $\bar{E}$. The first does e that is enabled in the initial state. The second can do $\bar{e}$, but only after doing b, and this in turn can happen only when *Tht* terminates.

▶ **Lemma 6.** *If φ is not SAT then all runs of $\mathbb{S}_\varphi$ start with e. In other words, $e \in first(w)$ for every full run w of $\mathbb{S}_\varphi$.*

Proof. Suppose w is a full run without e in $first(w)$. Since e is always possible until $\bar{e}$ is executed, we must have $\bar{e}$ on w. This must be preceded by b. So the first part, call it u, of w consists of synchronizations of variable processes V_i and $\bar{V}_i$ with *Tht*, without *Lmb* moving until *Tht* completes its run, that is until it does b. From the form of *Tht* it follows that the first part of u is a sequence $u_1 \ldots u_n$ where each u_i is either θ_i or $\bar{\theta}_i$. This defines a valuation v. At this stage, the variable processes corresponding to literals true in v are in their bottom states where the actions on these literals are possible. The other variable processes are in their top states where the action on the corresponding literals are impossible. So, since *Lmb* cannot move, *Tht* can get to action b iff v is a satisfying valuation. As φ is not SAT this is impossible. Hence, such w cannot exist. ◀

From the above proof we can actually see that $\bar{e}$ appears on the run iff φ is satisfiable. We do not need this fact in the rest of the argument, but it may be helpful to fix the intuitions.

▶ **Lemma 7.** *If φ is not SAT then the transition system from Figure 4 is a sound and complete transition system for $\mathbb{S}_\varphi$.*

Proof. First observe that in the state just before α's, all variable processes are in their bottom states, so all events x_i and $\bar{x}_i$ are enabled. Thus, all local paths of *Tht* on α's are feasible. Then action b is possible, but $\bar{e}$ is disabled since *Lmb* is in its bottom state. Thus, the transition system from Figure 4 is sound: every full path is a run of $\mathbb{S}_\varphi$.

It remains to verify that the transition system is complete. By Lemma 6, e is a first action of all full runs of $\mathbb{S}_\varphi$. Take such a run ew. Looking at $\mathbb{S}_\varphi$ we see that after e there are four possible actions: $\lambda_1, \bar{\lambda}_1, \theta_1, \bar{\theta}_1$. We will consider only the case when the next action is θ_1 as the argument for the other possibilities is analogous. So $w = \theta_1 u_1 \bar{\lambda}_1 u_1'$ for some sequences u_1 and u_1'; observe that $\bar{\lambda}_1$ must appear on the run, as after $e\theta_1$ there is no action that can disable it. Since the only action on which Lmb can synchronize is $\bar{\lambda}_1$, all actions in u_1 are synchronizations with Tht. Clearly they do not involve $\bar{V}_1$, as there is no way to execute $\bar{\theta}_1$ at this stage. Hence, u_1 is independent of $\bar{\lambda}_1$ giving us that w is trace equivalent to $\theta_1 \bar{\lambda}_1 u_1 u_1'$. Repeating this reasoning we obtain that w is trace equivalent to a run in the transition system from Figure 4. ◀

▶ **Lemma 8.** *If φ is SAT then there are runs containing $\bar{e}$. In every sound and complete reduced transition system for $\mathbb{S}_\varphi$ there are at least as many states as there are satisfying valuations for φ.*

Proof. For a valuation v we consider a run w_v taking θ_i if $v(x_i) = true$ and taking $\bar{\theta}_i$ if $v(x_i) = false$. For example, if x_1, and x_n hold but x_2 does not hold in v then this run would look something like:

$$\theta_1 \bar{\theta}_2 \dots \theta_n \alpha^1_{i_1} \dots \alpha^k_{i_k} b \bar{e} \bar{\lambda}_1 \lambda_2 \dots \bar{\lambda}_n$$

Here $\alpha^j_{i_j}$ is a literal that holds in the clause j. Since v is a satisfying valuation, process Tht can get to b.

Observe that there is no concurrency in the run w_v. All θ and $\bar{\theta}$ actions synchronize with process Tht. Then b and $\bar{e}$ are also dependent on each other as they happen on the same process. Action $\bar{e}$ is the first action of Lmb. It is followed by a sequence of actions of process Lmb.

Consider two different valuations $v_1 x$ and v_2 satisfying φ. Let s_1 be the state of system $\mathbb{S}_\varphi$ reached after $\bar{e}$ on the run w_{v_1}. Similarly, for s_2 and w_{v_2}. In state s_1, the variable processes that are in the bottom states are those corresponding to literals that are true in v_1 and similarly for v_2. Since v_1 and v_2 are distinct valuations, the states s_1 and s_2 are distinct. So any sound and complete transition system for $\mathbb{S}_\varphi$ must have at least as many states as there are satisfying valuations of φ. ◀

We write $|\varphi|$ for the length of φ. Observe that this is an upper bound on the number of variables as well as on the number of clauses in φ, namely, $n, k < |\varphi|$.

▶ **Corollary 9.** *If φ is not SAT then $minTS(\mathbb{S}_\varphi) \leq 6|\varphi|$ states. If φ is SAT then $minTS(\mathbb{S}_\varphi)$ is bigger than the number of satisfying valuations for φ.*

Now, we prove Theorem 5 for concurrent systems. Suppose to the contrary that Alg is an excellent POR algorithm. We use it to solve SAT in deterministic polynomial time.

Let $q(x)$ and $r(x)$ be the polynomials associated to Alg. Namely, Alg working in time $r(|\mathbb{S}| + |minTS(\mathbb{S})|)$ produces a sound and complete reduced transition system of size at most $q(|minTS(\mathbb{S})|)$.

Given a formula ψ, consider an integer m and a formula

$$\varphi \equiv \psi \wedge (z_1 \vee z_2) \wedge \dots \wedge (z_{2m-1} \vee z_{2m})$$

where $z_1, \dots, z_{2m}$ are new variables. Clearly φ is satisfiable iff ψ is. If ψ is satisfiable then φ has at least 2^m satisfying valuations.

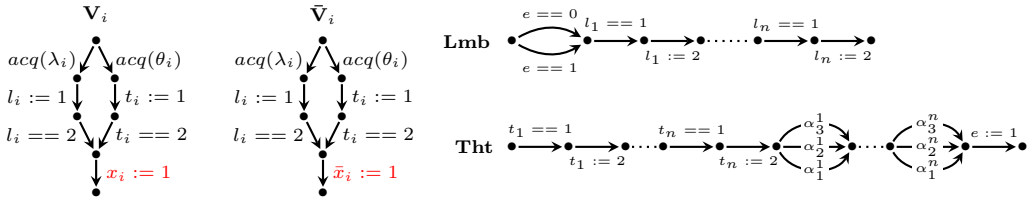


Figure 5 Processes V_i and $\bar{V}_i$ for propositional variable i, and special processes Lmb and Tht. Await instructions are written as equality checks, and writes as assignments. Instruction α_i^j tests if the corresponding variable is 1.

Now we construct our system $\mathbb{S}_\varphi$ and run Alg on it for $r(12|\varphi|)$ time. If φ is not SAT then, by Corollary 9, the algorithm stops and produces a sound and complete transition system. If φ is SAT then by Corollary 9 the algorithm cannot stop in this time as the smallest sound and complete transition system for $\mathbb{S}_\varphi$ has at least 2^m states, and we can choose m big enough so that $2^m > r(6|\varphi|)$.

4.2 Await and locks

In this section we show that the same phenomenon appears in concurrent programs. Here instead of direct synchronizations we have variables and locks. The only operations on variables are write and await.

The argument follows the same lines as the one for the direct synchronizations. Observe that the proof of Theorem 5 above uses Lemmas 6, 7 and 8. So it is enough to show that the three lemmas hold for concurrent programs with locks and variables to prove Theorem 5.

Given a propositional formula φ, we build a concurrent program $\mathbb{P}_\varphi$. As in the previous construction, we will have two processes for each propositional variable in φ, and two special processes: one for lambdas and one for thetas. However, we do not need processes E and $\bar{E}$ anymore. Initially all variables in $\mathbb{P}$ are set to 0.

Figure 5 shows the processes V_i and $\bar{V}_i$ for propositional variable i, as well as the two special processes Lmb and Tht. The processes V_i and $\bar{V}_i$ use two locks, λ_i and θ_i. After acquiring one of the two locks, each process sets the corresponding variable l_i or t_i to 1 in order to communicate their choice to the two special processes Tht and Lmb. Next, it awaits an acknowledgment from the special process in the form of the value of the variable l_i or t_i becoming 2. Finally, it sets x_i or $\bar{x}_i$ to 1.

The two special processes Lmb and Tht are quite similar to the previous construction. Only process Tht depends on the formula φ we are encoding. Process Lmb first tests if the value of the variable e is 0 or 1. Next it awaits for variable l_1 to become 1, and then sets it to 2. The same pattern repeats for l_2 up to l_n. Similarly, process Tht awaits for variable t_i to become 1, and then sets it to 2. After that, it tests if the formula is true, under the chosen valuation, and finally sets e to 1.

As in the previous section, we claim that if φ is not satisfiable, then there is a sound and complete transition system of a small size. This transition system has a similar structure to the one in the previous section (Figure 4), but is longer due to additional actions await and write. So our argument, while similar, becomes more complicated too.

▶ **Lemma 10.** *If φ is not satisfiable, then e can be set to 1 only after Lmb process executes $e == 0$.*

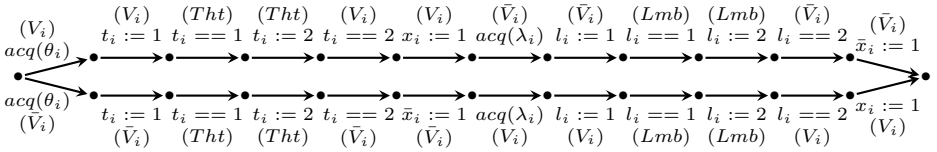


Figure 6 Block of actions in $\mathrm{TS}(\mathbb{P}_\varphi)$ when φ is not satisfiable. Await instructions are written as equality checks, and writes as assignments.

Proof. Suppose not. Then there is a run setting e to 1 without doing $e == 0$; meaning that e is set to 1 before Lmb moves. On this run all t_i must be set to 1 by processes V_i and $\bar{V}_i$, and then set to 2 by process Tht. Exactly one among processes V_i and $\bar{V}_i$ can proceed to setting x_i or $\bar{x}_i$ to 1; as the other needs to acquire λ_i, and then wait for l_i to become 2. Hence, when Tht reaches its α part, only one of x_i or $\bar{x}_i$ is set to 1. This defines a valuation, and Tht can set e to 1 if and only if this valuation satisfies φ. Hence, if φ is not satisfiable, then Tht cannot set e to 1. ◀

▶ **Lemma 11.** *If φ is not satisfiable, then there is a sound and complete transition system of size $O(n)$.*

We describe the transition system $\mathrm{TS}(\mathbb{P}_\varphi)$. By the previous lemma, if φ is not satisfiable, Lmb must perform $e == 0$ before e is set to 1. Thus, every run is equivalent to a run starting with $e == 0$. Then after the first transition, $\mathrm{TS}(\mathbb{P}_\varphi)$ is a sequence of blocks as shown in Figure 6. The upper line and the lower line depend on which process takes the θ_i lock and the λ_i lock. Then, these n blocks in $\mathrm{TS}(\mathbb{P}_\varphi)$, we have the α part of process Tht, including the last transition that sets variable e to 1. So the size of the transition system $\mathrm{TS}(\mathbb{P}_\varphi)$ is linear in the size of φ.

The proof of the third lemma is the same as for the previous construction.

▶ **Lemma 12.** *If φ is SAT then there are runs containing $e == 1$. In every sound and complete reduced transition system for $\mathbb{P}_\varphi$ there are at least as many states as there are satisfying valuations for φ.*

The three lemmas give us the proof of Theorem 5 for concurrent programs with locks and variables.

4.3 Just await

The previous construction uses locks and await instructions on variables. There is a number of classical algorithms for implementing locks with await, like Peterson, Bakery, or Szymanski. So it is natural to ask if we could eliminate locks from our construction. This is the goal of this section.

In the previous construction we used n locks, but each lock was shared between only two processes, and taken only once. We cannot just use one of the standard lock implementations out of the box because our construction is based on V_i and $\bar{V}_i$ competing for two locks. Moreover, we need to be careful with the number of states of the resulting transition system. So while there are no important new ideas in the encoding below, we face a challenge resembling a bit the classical problem of writing a program that prints itself.

Our solution is inspired by Peterson's mutual exclusion algorithm for 2 processes that uses a shared variable *turn* (T in Figure 2) to order the processes when they both request access to the critical section. Figure 7 (left) shows the two processes V_i and $\bar{V}_i$ that only use

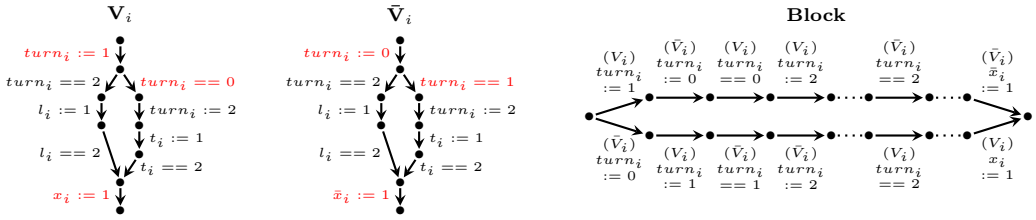


Figure 7 Processes V_i and $\bar{V}_i$ without locks for propositional variable i (left), and block of actions in TS($\mathbb{P}_\varphi$) when φ is not satisfiable (right). Await instructions are written as equality checks, and writes as assignments.

instructions await and write. Following the principle of Peterson's algorithm, the processes V_i and $\bar{V}_i$ share a variable $turn_i$ that replaces the lock operations, and that ensures that one of V_i and $\bar{V}_i$ will set t_i while the other will set l_i. The await instructions on variable $turn_i$ are followed by the same sequences of actions on t_i or l_i as previously. The two special processes *Lmb* and *Tht* are as in Figure 5. For notational convenience, we assume that the variables $turn_i$ are initialized to 2, whereas e is initialized to 0. The differences between V_i and $\bar{V}_i$ are highlighted in Figure 7.

The proof takes the same three steps as the previous one.

▶ **Lemma 13.** *If φ is not satisfiable, then e can be set to 1 only after Lmb process performs $e == 0$.*

Proof. Suppose there is a run where $e == 0$ is not a first action. This can only happen when $e := 1$ is executed before process *Lmb* starts. Hence, all t_i must be set to 1 by variable processes. This means that exactly one of the processes V_i or $\bar{V}_i$ can complete its operations and set x_i or $\bar{x}_i$ to 1; while the other will not pass $l_i == 2$. If φ is not satisfiable then this is not enough for *Tht* process to reach the point where it can set e to 1. ◀

▶ **Lemma 14.** *If φ is not satisfiable, then there is a sound and complete transition system of size $O(n)$.*

The argument is similar to the one for the construction with locks. TS($\mathbb{P}$) has a similar structure as before, with blocks as in Figure 7 (right).

▶ **Lemma 15.** *If φ is SAT then there are runs containing $e == 1$. In every sound and complete reduced transition system for $\mathbb{P}_\varphi$ there are at least as many states as there are satisfying valuations for φ.*

The third lemma is the same as in the previous construction, and its proof is also the same. The tree lemmas give us the proof of Theorem 5 for concurrent programs using only await operations.

5 An approach to partial-order reduction

In the light of the negative results from the previous section, it is clear that practical approaches to stateful partial-order reduction should be based on heuristics, and cannot aim at optimality, or even at approximate optimality up to a polynomial factor. In this section we propose a concrete problem, that when solved with a satisfactory heuristic should advance the quality of our partial-order methods. This approach also allows to make a link with stateless partial-order methods. It leads to a simple trace-optimal algorithm for the stateless case for a restricted class of concurrent systems.

We start with a simple algorithm for constructing trace-optimal reduced transition systems. It uses three ingredients that we describe in more details later.

- Lexicographic order on sequences allowing to determine a representative run for each class of a por-equivalence relation $\approx$.
- Sleep sets giving enough information about the exploration context.
- An oracle "includes first sets", denoted *IFS*, permitting to avoid sleep-blocked runs.

Lexicographic ordering has been already used in the context of partial-order reduction [19]. Sleep sets are one of the classical concepts for partial-order methods [14]. *IFS* oracle is a direct way of avoiding sleep-blocked executions, a well known challenge in partial-order reduction [1]. Our contribution here is to make the *IFS* problem explicit. Without surprise at this stage, the *IFS* problem is NP-hard, but we show one case where it is polynomial.

Let us fix a concurrent system $\mathbb{S}$. Let us also assume that we have some linear ordering on actions of $\mathbb{S}$. This determines a lexicographic ordering on sequences of actions. Given some equivalence relation on sequences we can use the lexicographic order to define representatives for equivalence classes of $\approx$ relation.

▶ **Definition 16.** *We say that w is a* lex-sequence *if w is the smallest lexicographically among all sequences equivalent to it: that is the smallest sequence in $\{v : v \approx w\}$. A* lex-run *of* TS($\mathbb{S}$) *is a run whose labels form a lex-sequence.*

We are going to present an algorithm enumerating full lex-runs of TS($\mathbb{S}$). For this we will use the "includes first set (IFS)" oracle. The idea is as follows. Suppose the algorithm has reached a state s and produced a sleep set $\mathit{sleep}(s)$ containing actions that need not be explored from s. We would like to check if there is something left to be explored. We need to check if the exploration is not *sleep-blocked*, namely, if there is a maximal run u from s with $\mathit{first}(u) \cap \mathit{sleep}(s) = \emptyset$. If there is such run, we need to explore one of the actions in $\mathit{first}(u)$ from s.

▶ **Definition 17** (IFS). *Let s be a state of* TS($\mathbb{S}$) *and B a subset of actions. We say that B* includes a first-set in *s if there is a maximal run u from s with $\mathit{first}(u) \subseteq B$. We write $\mathit{IFS}(s, B)$ when there exists such a maximal run u.*

Listing 1 presents a very simple algorithm enumerating all full lex-runs of TS($\mathbb{S}$). The algorithm also gives us an opportunity to explain sleep sets in details. Each node n of the tree constructed by the algorithm is a pair consisting of a state of TS($\mathbb{S}$), denoted $s(n)$, and a set of actions $\mathit{sleep}(n)$. For readability, we write $\mathit{enabled}(n)$ instead of $\mathit{enabled}(s(n))$, for the set of outgoing actions from $s(n)$.

Sleep sets are a very elegant mechanism to gather some information about the exploration context. They can be computed top-down when constructing an exploration graph. At the root, the sleep set is empty. For a node n and a transition $n \xrightarrow{e} n_e$ we have $\mathit{sleep}(n_e) = (\mathit{sleep}(n) \cup \{a_1, \ldots, a_k\}) - De$, where $a_1, \ldots, a_k$ are labels of transitions from n created before the e-transition; and the final $-De$ term means that we remove all the actions dependent on e. The intuition behind this formula is as follows. Assume that we keep an invariant:

> (*sleep-invariant*): after exploration of a node n for every maximal run u from $s(n)$ in TS such that $\mathit{first}(u) \cap \mathit{sleep}(n) = \emptyset$ we have a path v from n with $u \approx v$.

Then the formula for $\mathit{sleep}(n_e)$ says that we need not explore from n_e a run starting, say, with a_1 if $a_1 I e$. This is because when looking from n such a run has a_1 in its first set, and it has already been explored from a_1's successor of n.

Listing 1 Lex exploration with sleep sets, constructs a tree of maximal runs.

```
1  procedure main(𝕊):
2    create node n^0 with s(n^0) = s^0 and sleep(n^0) = ∅
3    TreeExplore(n^0)
4
5  procedure TreeExplore(n):
6    Sl := sleep(n) // invariant: Sl = sleep(n) ∪ {labels of transitions outgoing from n}
7    while enabled(n) − Sl ≠ ∅:
8        choose smallest e ∈ (enabled(n) − Sl) w.r.t. linear ordering on actions
9        let s' such that s(n) −e→ s' in TS(𝕊)
10       if IFS(s', enabled(s') − (Sl − De)):
11         create node n' with s(n') = s' and sleep(n') = Sl − De
12         add edge n −e→ n'
13         TreeExplore(n')
14       Sl := Sl ∪ {e}
```

The algorithm from Listing 1 examines all enabled transitions in a node n in our fixed order on actions. For every enabled transition $s(n) \xrightarrow{e} s'$ it uses the *IFS* oracle in Line 10 to decide if it is necessary to explore it. If the answer is positive, then it creates node n' with an appropriate sleep set. If not, then it skips the transition. This guarantees that the algorithm is trace-optimal.

▶ **Lemma 18.** *The algorithm in Listing 1 constructs a tree such that: (i) every full run in the tree is a full lex-run of* TS(𝕊)*, and (ii) for every full run u of* TS(𝕊) *there is a unique full run v in the tree with $v \sim u$.*

We could use this algorithm in stateless model-checking if we had an implementation of $IFS(s, B)$. In a stateful version, the algorithm is not very interesting as trees produced by this algorithm can be, and often are, orders of magnitude bigger than TS(𝕊), the transition system of 𝕊 without any reduction. But we can modify the algorithm to produce a graph instead of a tree. For this we need to introduce a subsumption relation between nodes.

▶ **Definition 19.** *We say that n* subsumes *n', in symbols $n \triangleleft n'$ if the states in n and n' are the same: $s(n) = s(n')$, and there are less sleep-blocked actions from n than from n': $sleep(n) \subseteq sleep(n')$.*

Observe that if n subsumes n', all runs that are not sleep blocked from n' are also not sleep blocked from n. Thus replacing n' by n still yields a sound and complete reduced transition system. Thus, we add subsumption to Listing 1 as shown in Listing 2. We first check if the successor s' of s is subsumed by an existing node in line 11. If yes, then, instead of creating a new node, we add an edge to the subsuming node (line 12). Otherwise, we proceed as in Listing 1. Observe that the algorithm in Listing 2 builds a directed acyclic graph (DAG) instead of a tree, hence avoiding visiting a node more than once.

The new algorithm does not satisfy a statement as in Lemma 18 because edges added due to the subsumption relation may create paths that are not lex-runs. We can get a variant of this optimality property when looking at states. Let us call a state *lex-useful* if it appears on a full lex-run.

▶ **Lemma 20.** *The algorithm with subsumption in Listing 2 builds a reduced transition system* TS_r *such that: (i) every full run in* TS_r *is a full lex-run of* TS(𝕊)*, and (ii) for every full run u of* TS(𝕊) *there is a full run v in* TS_r *with $v \sim u$. Moreover, for every node n, its state $s(n)$ is lex-useful.*

Listing 2 Lex exploration with sleep sets and subsumption, constructs a graph.

```
1  procedure main(𝕊):
2    create node n⁰ with s(n⁰) = s⁰ and sleep(n⁰) = ∅
3    Explored := ∅
4    DAGExplore(n⁰)
5
6  procedure DAGExplore(n):
7    Sl := sleep(n) // invariant: Sl = sleep(n) ∪ {labels of transitions outgoing from n}
8    while enabled(n) − Sl ≠ ∅:
9        choose smallest e ∈ (enabled(n) − Sl) w.r.t. linear ordering on actions
10       let s' such that s(n) −e→ s' in TS(𝕊)
11       if ∃n'' ∈ Explored such that n'' ◁ (s', Sl − De):
12         add edge n −e→ n''
13       else if IFS(s', enabled(s') − (Sl − De)):
14         create node n' with s(n') = s' and sleep(n') = Sl − De
15         add edge n −e→ n'
16         DAGExplore(n')
17       Sl := Sl ∪ {e}
18   add n to Explored
```

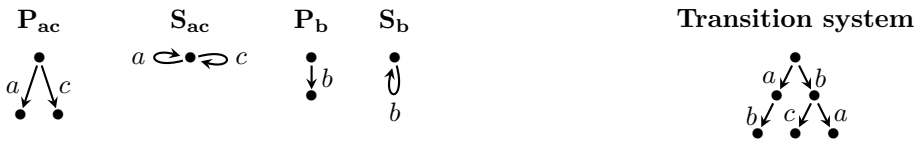


Figure 8 Sleep sets are needed for optimality.

▶ Remark 21. Sleep sets are needed for the optimality result, in a sense that some information about exploration context is needed. Consider the system of four process in Figure 8 (left) and its runs (right). The algorithm first explores the run ab, and then in the root state asks if there is a run where a is not a first action. The answer is positive because of the run bc. The algorithm then decides to explore b as it is the smallest enabled action. At this point it arrives at n_b. Without a sleep set it has no choice but to explore both a and c from n_b. But $ba \sim ab$ so this is not trace-optimal. With sleep sets we get $a \in sleep(n_b)$ which is exactly what is needed to block unnecessary exploration. So an idealized view suggested by Section 3 that partial-order reduction boils down to finding good covering source sets is not the whole story. Fortunately, the additional required information is easy to compute from the context.

5.1 The IFS oracle

The use of *IFS* queries gives a very clean approach to partial-order reduction, unfortunately the *IFS* problem is NP-hard.

▶ **Proposition 22.** *The following problem is* NP*-hard: given a concurrent system* $\mathbb{S}$*, its global state* s*, and a set of actions* B*, does* $IFS(s, B)$ *hold?*

Proof. Suppose we are given a 3CNF formula φ consisting of m clauses over variables $x_1, \dots, x_n$. Say the j-th clause is of the form $(\alpha_j^1 \vee \alpha_j^2 \vee \alpha_j^3)$ where each α is either some variable x_i or its negation $\bar{x}_i$. We will construct a concurrent program $\mathbb{P}_\varphi$ such that φ is satisfiable if and only if $IFS(s_0, B)$ holds for the initial state s_0 and B some set of actions we make precise below.

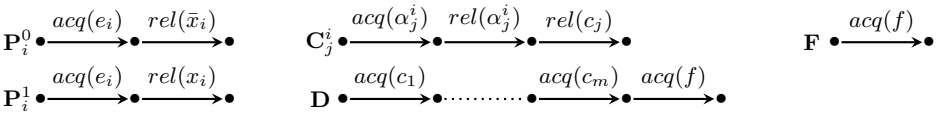


Figure 9 A concurrent program $\mathbb{P}_\varphi$ with locks encoding SAT(φ) for a 3CNF formula φ.

All actions in $\mathbb{P}_\varphi$ will implement taking or releasing a lock. There is one lock per propositional variable and its negation, $\{x_1, \ldots, x_n, \bar{x}_1, \ldots, \bar{x}_n\}$; one per clause, $\{c_1, \ldots, c_m\}$; one additional lock for every variable $\{e_1, \ldots, e_n\}$ in $\mathbb{P}_\varphi$, as well as a special lock f.

System $\mathbb{P}_\varphi$ is presented in Figure 9. For every propositional variable x_i there are two processes P_i^0 and P_i^1. For every clause $(\alpha_j^1 \vee \alpha_j^2 \vee \alpha_j^3)$ there are processes C_j^1, C_j^2, C_j^3. Additionally, there are two processes D and F.

Consider the state s where every process is in its initial state, and where the locks $\{x_1, \ldots, x_n, \bar{x}_1, \ldots, \bar{x}_n, c_1, \ldots, c_m\}$ are taken whereas the other locks are available. At this point the only processes enabled are F, and the processes corresponding to propositional variables, namely $P_1^0, P_1^1, \ldots, P_n^0, P_n^1$. Process F can take the lock f. For each i, either process P_i^0 or process P_i^1 takes the lock e_i which amounts to choosing a value for variable x_i, as either the lock x_i or the lock $\bar{x}_i$ is released, respectively. This may allow some process corresponding to a clause to move. For example, suppose x_i was released, and α_j^1, the first literal of the j-th clause, is x_i. Then process C_j^1 can take x_i and release x_i thus testing if x_i was available. If the lock was available, then it can release c_j. Releasing c_j intuitively means that the j-th clause is satisfied. Thus, all locks $c_1, \ldots, c_m$ can be released if and only if the formula φ is satisfied by the valuation determined by locks from $\{x_1, \ldots, x_n, \bar{x}_1, \ldots, \bar{x}_n\}$ that have been released. If all $c_1, \ldots, c_m$ are released then process D can take all these locks and then take f. So if φ is satisfiable then there is a maximal run of the system where $acq(f)$ is not a first action. If φ is not satisfiable then there is no way to release all the locks $c_1, \ldots, c_m$, so D cannot take f. In this case $acq(f)$ is in the first set of every maximal run because process F can always take f. This shows that $IFS(s, \Sigma - \{acq(f)\})$ holds iff φ is satisfiable (where Σ is the set of all actions in $\mathbb{P}_\varphi$). ◀

In our opinion it is worthwhile to study heuristics for the *IFS* problem. One possibility is to use existing static [14, 30, 34] or dynamic [12, 1, 18, 22] partial-order techniques to compute, for each node, a set of actions to visit to ensure completeness. Another intriguing approach consists in encoding $IFS(s, B)$ as a SAT problem, and then use a solver. This makes sense since the *IFS* problem is NP-hard. Designing heuristics that are efficient in practice is out of the scope of this paper. Still, we show that the *IFS* problem can actually be solved efficiently in a special case of non-blocking systems.

There is a simple setting where the *IFS* test can be done in polynomial time. Here the only operations are reads and writes, and programs are straight lines, in particular there are no conditionals. This case is indeed considered in the stateless model-checking literature [5, 22]. We generalize this situation to the case when we have variables but no operation on a variable can ever be blocked. So instead of just reads and writes we can also have operations like test-and-set, or fetch-and-add.

▶ **Definition 23.** *An* extended concurrent program *is a concurrent program $\mathbb{P}$ with arbitrary instructions on variables. It is* fully non-blocking *if in every state every first action of every process is enabled. It is* linear *if every process is a sequence of instructions, meaning there is no branching or conditionals.*

We use the term fully non-blocking to differentiate from non-blocking from [1] that is rather a forward non-blocking condition.

▶ **Proposition 24.** *If $\mathbb{P}$ is a linear, fully non-blocking extended concurrent program then IFS test can be done in polynomial time.*

Recall that $IFS(s, B)$ means that there is a maximal run u from s with $first(u) \subseteq B$. Let $\bar{B}$ denote the set of actions not in B. For a sequence v of actions we denote by $\bar{B}_v$ the set $\bar{B} - Dv$, namely the set of actions in $\bar{B}$ that are not dependent on actions in v. Then, we can check if $IFS(s, B)$ by constructing a run as follows. Suppose v is an already constructed run from s. Extend v with the smallest action enabled in the state reached after v, that is not in $\bar{B}_v$. When there is no such action return true if v is maximal, and false otherwise.

Thus, in the case covered by the above proposition the algorithm from Listing 1 is optimal and works in polynomial time w.r.t. size of the constructed tree. It works differently than race-reversal algorithms [1, 3].

The *IFS* test is NP-hard when one of the two conditions of Proposition 24 is lifted. This does not preclude that there is some case between fully non-blocking and blocking for which IFS is in Ptime. However, it is quite hard to imagine how to weaken the linearity condition.

6 Conclusions

Our main result is a negative one, indicating that we cannot hope for too much in terms of theoretically good partial-order algorithms. It has been known before that certain existing partial-order methods are NP-hard [30], but here we show that no method can be both polynomially close to optimal and not NP-hard.

As we have explained in the introduction, we believe that stateful partial-order methods have applications, and that there are even more applications to come. In this context our negative result gives us an argument for looking at well motivated heuristics rather than for a theoretically grounded result. We propose a new schema of a partial-order reduction algorithm based on IFS test. While in general the test is NP-complete, we hope that good heuristics for this test can translate into progress in partial-order methods. This is a direction we are actively investigating.

There is a big variety of concurrent systems, in particular depending on communication operations between processes. We have exhibited one case when IFS test can be done in Ptime. Intuitively, it captures "fully non-blocking" situations. An interesting question is what happens in the forward non-blocking case [1], or maybe to some other formalization of non-blocking in the spirit wait-free operations [16] or MPI [27].

References

1 Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. Source Sets: A Foundation for Optimal Dynamic Partial Order Reduction. *Journal of the ACM*, 64(4):1–49, 2017. doi:10.1145/3073408.

2 Parosh Aziz Abdulla, Mohamed Faouzi Atig, Frederik Meyer Bønneland, Sarbojit Das, Bengt Jonsson, Magnus Lang, and Konstantinos Sagonas. Tailoring Stateless Model Checking for Event-Driven Multi-threaded Programs. In Étienne André and Jun Sun, editors, *Automated Technology for Verification and Analysis*, pages 176–198. Springer Nature Switzerland, 2023. doi:10.1007/978-3-031-45332-8_9.

3 Parosh Aziz Abdulla, Mohamed Faouzi Atig, Sarbojit Das, Bengt Jonsson, and Konstantinos Sagonas. Parsimonious optimal dynamic partial order reduction. In Arie Gurfinkel and Vijay Ganesh, editors, *Computer Aided Verification - 36th International Conference, CAV 2024,*

Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part II, volume 14682 of *Lecture Notes in Computer Science*, pages 19–43. Springer, 2024. doi:10.1007/978-3-031-65630-9_2.

4 Parosh Aziz Abdulla, Mohamed Faouzi Atig, Sarbojit Das, Bengt Jonsson, and Konstantinos Sagonas. Trading space for simplicity in stateless model checking. In Susanne Graf, Paul Pettersson, and Bernhard Steffen, editors, *Real Time and Such - Essays Dedicated to Wang Yi to Celebrate His Scientific Career*, volume 15230 of *Lecture Notes in Computer Science*, pages 79–97. Springer, 2025. doi:10.1007/978-3-031-73751-0_8.

5 Pratyush Agarwal, Krishnendu Chatterjee, Shreya Pathak, Andreas Pavlogiannis, and Viktor Toman. Stateless Model Checking Under a Reads-Value-From Equivalence. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification*, volume 12759, pages 341–366. Springer International Publishing, 2021. doi:10.1007/978-3-030-81685-8_16.

6 Marek Chalupa, Krishnendu Chatterjee, Andreas Pavlogiannis, Nishant Sinha, and Kapil Vaidya. Data-centric dynamic partial order reduction. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–30, 2018. doi:10.1145/3158119.

7 Berk Cirisci, Constantin Enea, Azadeh Farzan, and Suha Orhun Mutluergil. A Pragmatic Approach to Stateful Partial Order Reduction. In Cezara Dragoi, Michael Emmi, and Jingbo Wang, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 13881, pages 129–154. Springer Nature Switzerland, 2023. doi:10.1007/978-3-031-24950-1_7.

8 Camille Coti, Laure Petrucci, César Rodríguez, and Marcelo Sousa. Quasi-optimal partial order reduction. *Formal Methods in System Design*, 57(1):3–33, 2021. doi:10.1007/s10703-020-00350-4.

9 Frank S. De Boer, Marcello Bonsangue, Einar Broch Johnsen, Violet Ka I Pun, S. Lizeth Tapia Tarifa, and Lars Tveito. SymPaths: Symbolic Execution Meets Partial Order Reduction. In Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, and Mattias Ulbrich, editors, *Deductive Software Verification: Future Perspectives*, volume 12345, pages 313–338. Springer International Publishing, 2020. doi:10.1007/978-3-030-64354-6_13.

10 Azadeh Farzan. Commutativity in Automated Verification. In *2023 38th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–7. IEEE, 2023. doi:10.1109/LICS56636.2023.10175734.

11 Azadeh Farzan, Dominik Klumpp, and Andreas Podelski. Sound sequentialization for concurrent program verification. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 506–521. ACM, 2022. doi:10.1145/3519939.3523727.

12 Cormac Flanagan and Patrice Godefroid. Dynamic Partial-Order Reduction for Model Checking Software. In *POPL'05*, 2005.

13 Patrice Godefroid. Using partial orders to improve automatic verification methods. In Edmund M. Clarke and Robert P. Kurshan, editors, *Computer-Aided Verification*, pages 176–185. Springer, 1991. doi:10.1007/BFb0023731.

14 Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems An Approach to the State-Explosion Problem*. PhD thesis, Université de Liège, 1994.

15 R. Govind, Frédéric Herbreteau, Srivathsan, and Igor Walukiewicz. Abstractions for the local-time semantics of timed automata: A foundation for partial-order methods. In *Proceedings of the 37th Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 1–14. ACM, 2022. doi:10.1145/3531130.3533343.

16 Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991. doi:10.1145/114005.102808.

17 Casper S. Jensen, Anders Moller, Veselin Raychev, Dimitar Dimitrov, and Martin Vechev. Stateless model checking of event-driven applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 57–73. Association for Computing Machinery, 2015. doi:10.1145/2814270.2814282.

18 Bengt Jonsson, Magnus Lang, and Konstantinos Sagonas. Awaiting for Godot Stateless Model Checking that Avoids Executions where Nothing Happens. In *{22nd Formal Methods in Computer-Aided Design, {FMCAD} 2022*. IEEE, 2022.

19 Vineet Kahlon, Chao Wang, and Aarti Gupta. Monotonic Partial Order Reduction: An Optimal Symbolic Partial Order Reduction Technique. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification*, volume 5643, pages 398–413. Springer Berlin Heidelberg, 2009. doi:10.1007/978-3-642-02658-4_31.

20 Shmuel Katz and Doron Peled. Verification of distributed programs using representative interleaving sequences. *Distributed Computing*, 6(2):107–120, 1992. doi:10.1007/BF02252682.

21 Michalis Kokologiannakis, Rupak Majumdar, and Viktor Vafeiadis. Enhancing GenMC's Usability and Performance. In Bernd Finkbeiner and Laura Kovács, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 66–84. Springer Nature Switzerland, 2024. doi:10.1007/978-3-031-57249-4_4.

22 Michalis Kokologiannakis, Iason Marmanis, Vladimir Gladstein, and Viktor Vafeiadis. Truly stateless, optimal dynamic partial order reduction. *Proceedings of the ACM on Programming Languages*, 6(POPL):1–28, 2022. doi:10.1145/3498711.

23 Michalis Kokologiannakis, Iason Marmanis, and Viktor Vafeiadis. Unblocking Dynamic Partial Order Reduction. In Constantin Enea and Akash Lal, editors, *Computer Aided Verification*, volume 13964, pages 230–250. Springer Nature Switzerland, 2023. doi:10.1007/978-3-031-37706-8_12.

24 Michalis Kokologiannakis, Iason Marmanis, and Viktor Vafeiadis. SPORE: combining symmetry and partial order reduction. *Proc. ACM Program. Lang.*, 8(PLDI):1781–1803, 2024. doi:10.1145/3656449.

25 Pallavi Maiya, Rahul Gupta, Aditya Kanade, and Rupak Majumdar. Partial Order Reduction for Event-Driven Multi-threaded Programs. In Marsha Chechik and Jean-François Raskin, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 680–697. Springer, 2016. doi:10.1007/978-3-662-49674-9_44.

26 Antoni W. Mazurkiewicz. Introduction to trace theory. In Volker Diekert and Grzegorz Rozenberg, editors, *The Book of Traces*, pages 3–41. World Scientific, 1995. doi:10.1142/9789814261456_0001.

27 Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.1*, November 2023. URL: https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf.

28 Burcu Kulahcioglu Ozkan, Rupak Majumdar, and Simin Oraee. Trace aware random testing for distributed systems. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–29, 2019. doi:10.1145/3360606.

29 Doron Peled. Partial-order reduction. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 173–190. Springer, 2018. doi:10.1007/978-3-319-10575-8_6.

30 Doron A. Peled. All from one, one for all: on model checking using representatives. In Costas Courcoubetis, editor, *Computer Aided Verification, 5th International Conference, CAV '93, Elounda, Greece, June 28 - July 1, 1993, Proceedings*, volume 697 of *Lecture Notes in Computer Science*, pages 409–423. Springer, 1993. doi:10.1007/3-540-56922-7_34.

31 The Anh Pham. *Efficient State-Space Exploration for Asynchronous Distributed Programs: Adapting Unfolding-Based Dynamic Partial Order Reduction to MPI Programs*. These de doctorat, Rennes, Ecole normale superieure, 2019.

32 César Rodríguez, Marcelo Sousa, Subodh Sharma, and Daniel Kroening. Unfolding-based Partial Order Reduction. *LIPIcs, Volume 42, CONCUR 2015*, 42:456–469, 2015. doi:10.4230/LIPICS.CONCUR.2015.456.

33 Samira Tasharofi, Rajesh K. Karmani, Steven Lauterburg, Axel Legay, Darko Marinov, and Gul Agha. TransDPOR: A Novel Dynamic Partial-Order Reduction Technique for Testing Actor Programs. In Holger Giese and Grigore Rosu, editors, *Formal Techniques for Distributed Systems*, pages 219–234. Springer, 2012. doi:10.1007/978-3-642-30793-5_14.

34 Antti Valmari. Stubborn sets for reduced state space generation. In G. Goos, J. Hartmanis, D. Barstow, W. Brauer, P. Brinch Hansen, D. Gries, D. Luckham, C. Moler, A. Pnueli, G. Seegmüller, J. Stoer, N. Wirth, and Grzegorz Rozenberg, editors, *Advances in Petri Nets 1990*, volume 483, pages 491–515. Springer Berlin Heidelberg, 1991. doi:10.1007/3-540-53863-1_36.

35 Bjorn Wachter, Daniel Kroening, and Joel Ouaknine. Verifying multi-threaded software with impact. In *2013 Formal Methods in Computer-Aided Design*, pages 210–217. IEEE, 2013. doi:10.1109/FMCAD.2013.6679412.

36 Chao Wang, Zijiang Yang, Vineet Kahlon, and Aarti Gupta. Peephole Partial Order Reduction. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963, pages 382–396. Springer Berlin Heidelberg, 2008. doi:10.1007/978-3-540-78800-3_29.

37 Yu Yang, Xiaofang Chen, Ganesh Gopalakrishnan, and Robert M. Kirby. Efficient Stateful Dynamic Partial Order Reduction. In Klaus Havelund, Rupak Majumdar, and Jens Palsberg, editors, *Model Checking Software*, pages 288–305. Springer, 2008. doi:10.1007/978-3-540-85114-1_20.

First-Order Store and Visibility in Name-Passing Calculi

Daniel Hirschkoff
LIP, ENS de Lyon, France

Iwan Quémerais
LIP, ENS de Lyon, France

Davide Sangiorgi
Università di Bologna, Italy
INRIA, Sophia Antipolis, France

Abstract

The π-calculus is the paradigmatical name-passing calculus. While being purely name-passing, it allows the representation of higher-order functions and store.

We study how π-calculus processes can be controlled so that computations can only involve storage of first-order values. The discipline is enforced by a type system that is based on the notion of visibility, coming from game semantics. We discuss the impact of visibility on the behavioural theory. We propose characterisations of may-testing and barbed equivalence, based on (variants of) trace equivalence and labelled bisimilarity, in the case where computation is sequential, and in the case where computation is well-bracketed.

2012 ACM Subject Classification Theory of computation → Process calculi

Keywords and phrases process calculi, behavioural equivalence, type system

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2025.23

Related Version *Extended Version*: `arXiv.2504.17350` [4]

Funding Work partly supported by the MIUR-PRIN project "Resource Awareness in Programming: Algebra, Rewriting, and Analysis" (RAP, ID P2022HXNSC).

Acknowledgements We have benefited from discussions with G. Jaber and E. Prebet.

1 Introduction

The π-calculus is one of the best known process calculi. It has a handful of operators – parallelism, input, output, restriction being the main ones – with which it achieves an amazing expressive power. Yet, it has a rich algebraic theory and a wide spectrum of proof techniques. In the π-calculus, computation is communication, by means of message-passing synchronisations between processes. Names are the communication units; new names may be created, and may themselves be exchanged in communications. When used for modelling, π-calculus is normally typed, so to be able to support structured data values, beginning with basic first-order values such as integers and booleans.

The π-calculus has been advocated as a model to interpret, and give semantics to, languages with higher-order features. For instance, the representation of functions as π-calculus processes has been widely studied, beginning with Milner's seminal work [16]. Usually, higher-order languages include forms of *store*. It is therefore important to understand how expected semantic properties of store can be captured in a language for pure concurrency such as the π-calculus.

An aspect of store with strong semantic consequences is the distinction between higher-order and first-order store. The term "first-order store" refers to a memory model where variables can hold only basic data types such as integers and booleans. This contrasts with a

© Daniel Hirschkoff, Iwan Quémerais, and Davide Sangiorgi;
licensed under Creative Commons License CC-BY 4.0

higher-order store, where variables can also store functions, or code, or references to these. Limiting the store to first-order data makes it easier to reason about program behaviours, and verify behavioural properties. It may also enhance security and predictability guarantees by preventing unintended behaviours associated with dynamic function storage and code injection. A striking example of a property that may be broken with higher-order store is termination. For instance, the simply-typed λ-calculus is strongly normalising and therefore terminating. This property is maintained by the addition of first-order store, but not with higher-order store, as functions may recursively reference and invoke each other through the store. In game semantics of higher-order languages, the distinction between first-order and higher-order store has led to studying the concept of *visibility*. Intuitively, visibility ensures us that each move in a computational game is justified by prior interactions. With the introduction of higher-order store, the visibility condition has to be relaxed, reflecting the added complexity in tracking information flow within a program.

The goal of this paper is precisely to understand the meaning of first-order store and visibility within the π-calculus. In functional languages based on the λ-calculus, imposing first-order store is simply determined by the constraint that all references may only store first-order values. In the π-calculus, in contrast, there is no explicit notion of reference, and therefore the meaning of "first-order store" requires some care. In π-calculi, a piece of store is usually represented by an output message. For instance $\overline{h}\langle 5\rangle$ may be thought of as the statement that name h currently contains 5. A process that inputs at h and then re-emits a message at h, such as $h(z).\,\overline{h}\langle z+1\rangle$, is a process that acts on such a store. In this case, h is a first-order name, hence h implements a piece of first-order store. An example of higher-order store is given by the process

$$P \stackrel{\text{def}}{=} a(b).\,((\overline{b}(d).\,!d.\,R) \mid c.\,\overline{b}(d').\,d'.\,R') \tag{1}$$

Here, $\overline{b}(d)$ is a bound output, and represents the output of a "new" name d, and similarly for $\overline{b}(d')$; whereas, e.g., $d.\,R$ and $c.\,Q$ (where Q is $\overline{b}(d').\,d'.\,R'$) represent inputs in which no value is exchanged (i.e., d and c carry `unit` values). In P, name a is higher-order, in that a carries a (pointer to a) service. In the input at a, the service name received in the input (the parameter b) is used after the input, but it is also stored, and used later when c is invoked. In other words, the visibility of c (the services that c may call when invoked) depends on interactions that have occurred earlier, at a. For this reason, P implements a form of higher-order store. When this does not happen, that is, the visibility of a higher-order name (the set of services that may be called when the name is invoked) may be determined at the point in which the name is created, the process implements a form of first-order store. (The word "service" is freely used, here and elsewhere, to indicate the input end of a name; in π-calculus, a name may have multiple input occurrences, in arbitrary syntactic positions within a process, in contrast with, say, the function declarations in functional languages, which are supposed to be unique.)

The amazing expressive power of the π-calculus also makes the contexts of the calculus very discriminating; as a consequence, behavioural equivalences, which are supposed to be preserved by the contexts of the calculus, are rather demanding relations. A well-established way of imposing constraints on the set of legal contexts, thus obtaining usefully coarser behavioural equivalences, is to adopt behavioural type systems. Types may be used in order to capture communication patterns on the usage of names, such as capabilities, linearity, sessions, and so on, e.g., [19, 11, 6, 1]. Type systems have also been designed to capture specific properties of processes, such as termination, deadlock-freedom, lock-freedom, e.g., [24, 12, 18].

A further step is then to tune the proof techniques of the π-calculus to such type systems, so to be able to actually prove the behavioural equalities that only hold in presence of types. For instance, in the case of may testing (or contextual equivalence) this may yield to refinements or modifications of the notion of trace equivalence; whereas in the case of barbed equivalence this may lead to refinements or modifications of the standard notion of labelled bisimilarity. The resulting proof techniques must be sound with respect to initial contextually-defined relations; ideally, they should also be complete.

In this paper we propose a type system that guarantees first-order store in π-calculus processes. The type system makes use of *visibility functions*, that is, partial finite functions that specify, for each higher-order name b that may occur free in a process, the set of services (i.e., higher-order names) that may be invoked when a call at b is made. In order to better understand the behavioural effects of visibility, we combine it first with *sequentiality* and then with *well-bracketing*. "Sequentiality" intuitively indicates the existence of a single thread of computation. That is, at any point there is a single process that is active and decides what the next computation step can be. In the encodings of the λ-calculus [16, 21], a process is active, i.e., it carries the thread, when it contains an unguarded output at a higher-order name. When encoding λ-calculi with first-order references, an active process may also expose the thread by exhibiting an input at a first-order name. We follow such a convention also in the paper: higher-order names carry the thread via outputs, first-order names via inputs (other choices technically would however be possible). Sequentiality allows us to expose, in isolation, the causality properties of service calls, within a standard (interleaving) semantics. These aspects would be partially or totally obscured by the presence of other concurrent threads.

In higher-order languages, the effects of having first-order store are particularly strong in absence of control operators; that is, using a terminology borrowed from game semantics, in a "well-bracketed" setting, where the call-return interaction behaviour between a term and its context follows a stack discipline. We therefore also consider the refinement of our visibility-based type system under the well-bracketing hypothesis. To see an example of the effect of visibility, consider the following well-bracketed example:

$$\boldsymbol{\nu} h\,(\,\overline{h}\langle 0\rangle \;\mid\; \overline{a}(c,q).\,(!c.\,h(z).\,\overline{h}\langle z+1\rangle \mid q(y).\,Q)\,)$$

Here, an external service a is called, exporting a local service c capable of incrementing a private reference h, and with a return channel q. If visibility holds, then the external environment cannot store c. Therefore, after providing an answer at q , the environment is unable to call c again, thus incrementing h, unless an access to c is explicitly provided again (by process Q). This property may be used to perform optimisations, possibly including garbage-collection of the service c if at some point c is not used anymore in Q.

We then study characterisations for both may testing and barbed equivalence, as special forms of trace equivalence and of labelled bisimilarity. Following the tradition of testing, the language includes special success signals, that are used by testers (i.e, processes running in parallel with the testees), and may be used at any time, to report success.

When developing such characterisations, some care is necessary so to distinguish between the visibility of the tested and of the tester processes, as they need not be the same. For this, we introduce visibility-constrained LTSs, in which process transitions are constrained by a visibility function, intuitively expressing the possibility for processes to perform a certain transition given a certain visibility function for the external observer. As a consequence of the transition, the visibility function may need to be updated, so to yield the visibility function for the next transition. In the case of well-bracketing, the constraints given by visibility have

to be coupled with those given by the stack names used to track well-bracketing (and again, a distinction has to be made between the stacks for the testers and those for the testees). Based on these LTSs, we introduce forms of trace equivalence and of labelled bisimilarity that are proved to be sound and complete, for may testing and barbed equivalence, both in the case of sequentiality and in that of well-bracketing.

We present the version of the π-calculus with higher-order and first-order names in Section 2. In Section 3, we define the type system implementing visibility for sequential processes. We define typed labelled transitions and present labelled characterisations of behavioural equivalences in Section 4. Section 5 is devoted to the extension with well-bracketing. For lack of space, the results about labelled bisimilarities are presented in [4].

2 Background: the π-Calculus Language

The π-calculus language that will be used is called, technically, Internal π (πI) [22] (all channels exchanged are fresh – no free object outputs), with the addition of *first-order values* (such as integers and booleans), and expressions that evaluate to first-order values, called *first-order expressions*. We do not specify what exactly first-order values and first-order expressions are. We simply assume that, in a closed process, a first-order expression evaluates to a first-order value. Moreover, only the output capability of names may be exported. This means that, for instance, in an input $a(b).P$ name b may only be used in output in P. This constraint is often followed in theory and applications of the π-calculus, as it simplifies various technical matters.

We assume that names are distinguished (partitioned) into *higher-order* names and *first-order* names, and *success names*. Higher-order names are the ordinary πI names; first-order names may only carry first-order values, and they may not be passed around (the second constraint simplifies the technical details, though it is not mandatory, see Section 3). In later sections, first-order names will be used to represent references. The fact that these names may only carry first-order values, together with constraints on visibility, will ensure that the calculus does not have (implicit or explicit forms of) higher-order store. The *success* names are used, in the tradition of testing equivalences, by tester processes so to report results of experiments on tested processes (thus, the processes compared in the examples of the paper do not contain success names).

The calculi in the paper will be typed. For simplicity we define our type systems as refinements of the most basic type system for π-calculus, namely Milner's *sorting* [15], in which names are partitioned into a collection of *types* (or sorts), and a sorting function maps types onto types. We assume that there is a sorting system under which all processes we manipulate are well-typed. Thus, the sorting system separates higher-order and first-order names; moreover we assume that first-order names are monadic, and that higher-order names carry a pair of a first-order value and a higher-order name. Thus $\overline{a}\langle v\rangle(b).P$ is the output of the first-order value v and of the fresh higher-order name b at the (higher-order) name a, with continuation P. When writing examples, we allow different sortings; for instance, writing $a.P$ and $\overline{b}.Q$ for inputs and outputs in which no value is exchanged.

The syntactic categories of the calculus are presented in Table 1. As usual, input and restriction are binding constructs, and give rise to the expected definitions of free and bound names. An expression or a process is *closed* if it does not have free (first-order) variables. It is intended that transitions are defined on closed processes.

The definition of structural congruence, written $\equiv$, and of the strong and weak (untyped) labelled transitions, written $\xrightarrow{\mu}$, $\Longrightarrow$, and $\stackrel{\hat{\mu}}{\Longrightarrow}$, are standard and are given in Appendix A (later we will introduce typed variants of the LTS). We note $\mathtt{fn}(P)$ (resp. $\mathtt{fn}(\mu)$) the set of free names of P (resp. μ). We sometimes abbreviate reductions $P \xrightarrow{\tau} P'$ as $P \longrightarrow P'$.

Table 1 The syntax of the calculus.

$$\begin{array}{rrrrr} [\textit{Higher-order names, } \texttt{HO}] & a,b,c & & [\textit{First-order expressions}] & e,f \\ [\textit{First-order names, } \texttt{FO}] & h,k & & [\textit{First-order variables}] & x,y \\ [\textit{Success names}] & \omega & & [\textit{First-order values}] & v,w \end{array}$$

$$\begin{array}{rrcl} [\textit{Inputs}] & \alpha & ::= & a(x,b) \mid h(x) \\ [\textit{Outputs}] & \overline{\alpha} & ::= & \overline{a}\langle e\rangle(b) \mid \overline{h}\langle e\rangle \mid \omega \\ [\textit{Processes}] & P,Q,R & ::= & \alpha.P \mid \overline{\alpha}.P \mid !\alpha.P \mid P \mid Q \mid \boldsymbol{\nu} a\, P \mid \boldsymbol{\nu} h\, P \\ & & & \mid \mathbf{0} \mid P+Q \mid \texttt{if } e=f \texttt{ then } P \texttt{ else } Q \end{array}$$

As behavioural equivalences we consider both may-testing and barbed equivalence. Both relations are contextual, in that they are defined by requiring a closure under all "observers". They both use "barbs" (or "success") signals; barbed equivalence, in addition, makes use of a bisimulation game on reductions. We write $\Downarrow_\omega$ to indicate the possibility of exhibiting an ω barb, possibly after some reductions; i.e., $P \Downarrow_\omega$ holds if there is P' with $P \Longrightarrow P'$ and P' has an unguarded occurrence of ω. Thus intuitively two processes are behaviourally equal if they can produce "the same barbs" under all legal observers running in parallel with them. As we are in a typed setting, observer and tested processes should have compatible typings. The meaning of "compatible" depends on the specific type system. For visibility, the meaning will be specified in Sections 3 and 5. We write $\Delta \vdash P$, and say that P is a Δ-*process*, if P is typable under Δ, and $\Delta \vdash P, Q$ if both $\Delta \vdash P$ and $\Delta \vdash Q$ hold. Behavioural equivalences are defined on closed processes; they are extended to open processes (i.e., processes with free first-order variables) in the usual manner, by considering all possible closing substitutions. We only consider *weak* behavioural equivalences (i.e., equivalences that abstract from internal moves of processes, the reductions), as these are, in practice, the most useful ones.

▶ **Definition 1** (May testing)**.** Suppose that Γ, Δ are compatible. Then two Δ-processes P and Q are *may testing equivalent at* Γ, written $P \cong^{\Gamma}_{\Delta} Q$, if for all Γ-process R and success name ω, we have $(R \mid P) \Downarrow_\omega$ iff $(R \mid Q) \Downarrow_\omega$.

▶ **Definition 2** (Barbed bisimilarity and equivalence)**.** *Barbed bisimilarity* is the largest symmetric relation $\dot{\approx}$ s.t. $P \dot{\approx} Q$ implies:

1. whenever $P \longrightarrow P'$, there exists Q' such that $Q \Longrightarrow Q'$ and $P' \dot{\approx} Q'$;
2. for each ω, if $P \Downarrow_\omega$ then also $Q \Downarrow_\omega$.

Suppose that Γ, Δ are compatible. Then two Δ-processes P and Q are *barbed equivalent at* Γ, written $P \approx^{\Gamma}_{\Delta} Q$, if for all R with $\Gamma \vdash R$ it holds that $P \mid R \dot{\approx} Q \mid R$.

We write $P \cong Q$ (resp. $P \approx Q$) when $P \cong^{\Gamma}_{\Delta} Q$ (resp. $P \approx^{\Gamma}_{\Delta} Q$) holds under any typing Δ for P, Q and any Γ compatible with Δ.

We now introduce some notations that will be useful below. In the paper we use partial functions whose codomains are finite powersets. If Δ is such a function, then $\Delta; i \mapsto A$ is the extension of Δ in which i is mapped onto A, assuming that i is not in $\texttt{dom}(\Delta) \cup \texttt{codom}(\Delta)$; conversely Δ^{-i} is the restriction of Δ in which the entry for i is erased, both in the domain and codomain of Δ. We also write $i \mapsto A, j$ as an abbreviation for $i \mapsto (A \cup \{j\})$; and $\Delta \lfloor i$ (resp. $\Delta \not\lfloor i$) means that Δ is (resp. is not) defined on i. Function $\Delta_1 \cap \Delta_2$ has domain $\texttt{dom}(\Delta_1) \cap \texttt{dom}(\Delta_2)$, and then is defined as $(\Delta_1 \cap \Delta_2)(p) = \Delta_1(p) \cap \Delta_2(p)$; and likewise for $\Delta_1 \cup \Delta_2$. We write $\Delta_1 \subseteq \Delta_2$ if $\texttt{dom}(\Delta_1) \subseteq \texttt{dom}(\Delta_2)$, and for each $i \in \texttt{dom}(\Delta_1)$, we have $\Delta_1(i) \subseteq \Delta_1(i)$.

3 A Type System for Visibility

Our type system for visibility is built on top of that for sequentiality [3]. While visibility could be defined as a standalone notion, the combination with sequentiality makes the behavioural effects sharper and easier to grasp.

Intuitively, sequentiality ensures us that at any time at most one interaction can occur; i.e., there is a single computation thread. A process that holds the thread, called *active*, decides what the next interaction can be. It does so by offering a single particle (input or output) that controls the thread. A given name may exercise the control on the thread either in output or in input. Following [3], we require that, in higher-order names, the thread is carried by outputs, whereas, in first-order names, it is carried by inputs. An input that carries the thread maintains the thread, whereas an output releases it. (These appear to be, practically, the most interesting combinations; other combinations could be adopted, with the expected modifications.) Thus, for instance, the following process P is well-typed:

$$P \stackrel{\text{def}}{=} \big(h(x).\,\overline{a}\langle x\rangle(b).\,(b(z,c').\,Q \mid \overline{h}\langle x\rangle)\big) \;\mid\; \overline{k}\langle v\rangle$$

The initial particles in P are the input at h and the output at k; however only the input at h carries the thread, as h and k are first-order. Underneath the input, the thread is maintained and released in the output at a (the thread will be acquired by the process in the external environment performing the input at a); later, when the input at b is performed, process Q obtains the thread back. Sequentiality however does not exclude non-determinism: e.g., an output particle may have the possibility of interacting with different inputs.

The type system for *visibility* intuitively allows us to specify which possible higher-order outputs can be performed by a process that becomes active. Type environments are partial functions, called *visibility functions*, acting on higher-order names and, for active processes, on the special token $*$, representing the active thread. (Later, when studying well-bracketing, type environments will have both a visibility function and a name stack.)

▶ **Definition 3** (Visibility function)**.** A *visibility function* is a finite partial function $\Delta : \mathtt{HO} \cup \{*\} \to \mathcal{P}_{\mathtt{fin}}(\mathtt{HO})$. Moreover the function should be *transitive*, that is, for all $o \in \mathtt{dom}(\Delta)$, if $\Delta \downharpoonright a$ and $a \in \Delta(o)$, then $\Delta(a) \subseteq \Delta(o)$.

To see the use of $*$, suppose $\Delta(*) = \{a, b\}$; this means that a Δ-process is active, and may only perform outputs at a and b, as in the following process:

$$\overline{a}\langle v\rangle(c).\,c(d).\,\overline{b}(w).\,\mathbf{0}$$

Name c is created in the output at a and therefore c inherits a's visibility.

Similarly, if $\Delta(d) = \{a, b\}$ then, in a Δ-process $d(x, c).\,P$, the continuation P of the input at d may only perform outputs at a, b or at c (as c is the higher-order name received in the input). The use of visibility functions also allows us to rule out implicit forms of higher order store; in the following example, p, q are higher order names used for continuations.

When extending Δ, we impose in $\Delta; a \mapsto A$ that $a \notin \mathtt{dom}(\Delta) \cup \mathtt{codom}(\Delta)$, that is, a is fresh for Δ. Similarly, $\Delta; * \mapsto A$ is defined only when $* \notin \mathtt{dom}(\Delta)$.

▶ **Example 4** (Implicit forms of higher-order store)**.** In the λ-calculus, if a term without higher-order references is well-bracketed then it necessarily obeys visibility. In our calculus, even though we do not have explicit higher-order references, we can simulate higher-order store. Consider

$$P \stackrel{\text{def}}{=} \overline{a}(b, p).\,b(n, c, q).\,\overline{q}\langle n\rangle.\,p(m).\,\overline{c}.\,\mathbf{0}$$

$$\texttt{O-HO}\ \frac{a \in \Delta(*) \quad \Delta^{-*}; b \mapsto \Delta(*), b \vdash P}{\Delta \vdash \overline{a}\langle e\rangle(b).\, P} \qquad \texttt{I-HO}\ \frac{\Delta; * \mapsto \Delta(a), b \vdash P}{\Delta \vdash a(x,b).\, P}$$

$$\texttt{O-FO}\ \frac{\Delta \vdash P \quad \Delta \not\downarrow *}{\Delta \vdash \overline{h}\langle e\rangle.\, P} \qquad \texttt{I-FO}\ \frac{\Delta \vdash P \quad \Delta \downarrow *}{\Delta \vdash h(x).\, P} \qquad \texttt{REP}\ \frac{\Delta \vdash \alpha.\, P \quad \Delta \not\downarrow *}{\Delta \vdash !\alpha.\, P} \qquad \texttt{SUCC}\ \frac{}{\Delta \vdash \omega.\, P}$$

$$\texttt{RES-HO}\ \frac{\Delta \vdash P}{\Delta^{-a} \vdash \nu a P} \qquad \texttt{RES-FO}\ \frac{\Delta \vdash P}{\Delta \vdash \nu h P} \qquad \texttt{NIL}\ \frac{\Delta \not\downarrow *}{\Delta \vdash \mathbf{0}} \qquad \texttt{SUM}\ \frac{\Delta \vdash P_1 \qquad \Delta \vdash P_2}{\Delta \vdash P_1 + P_2}$$

$$\texttt{PAR}\ \frac{\Delta_1 \vdash P_1 \qquad \Delta_2 \vdash P_2 \qquad (\Delta_1, \Delta_2) \in \texttt{split}(\Delta)}{\Delta \vdash P_1 \mid P_2} \qquad \texttt{IF}\ \frac{\Delta \vdash P_1 \quad \Delta \vdash P_2}{\Delta \vdash \texttt{if}\ e = f\ \texttt{then}\ P_1\ \texttt{else}\ P_2}$$

Figure 1 Typing rules for visibility.

In P the visibility $\Delta(p)$ is defined when the output at a is made, therefore $c \notin \Delta(p)$, and the final output at c breaks visibility; indeed, the input at b, which justifies the output at c, is answered by the output at q before the output at c.

Transitivity ensures us that the visibility functions, viewed as "named sets", represent transitive sets. Transitivity is needed to guarantee the behavioural expectations of visibility. If a process P calls a service a, with actual (higher-order) parameter b, then an output triggered by such a call (and possibly directed back to P) should be either at b or at a name in the visibility of a; however, the server a might call an external service c; this call should not allow an increase of the visibility associated to a; hence the transitivity constraint that the visibility of c is contained in that of a. Technically, transitivity is necessary to ensure preservation of typing under reduction (see Theorem 10 and Example 11).

▶ **Remark 5** (Transitivity and functional languages)**.** The issue of transitivity does not arise in the semantics of purely functional languages such as the λ-calculus [7]: visibility functions do appear, but the domain and codomain of such functions are disjoint (technically, in the game semantics terminology, the domain contains names introduced by Player, whereas the codomain contains names introduced by Opponent). This makes transitivity trivial.

In the typing of a parallel composition, the visibility function has to be split between the two components of the composition. This is important for elements of the domain of the function that are linear, and therefore may only appear in one, but not both, components; this is the case for $*$, and the linear continuation names that will be introduced in Section 5.

▶ **Definition 6** (Split)**.** Let Δ be a visibility function, we define $\texttt{split}(\Delta)$ as the set of pairs (Δ_1, Δ_2) such that $\Delta_1 \cup \Delta_2 = \Delta$ and $* \notin \texttt{dom}(\Delta_1) \cap \texttt{dom}(\Delta_2)$.

The typing rules are given in Table 1. We comment on a few of them. Rule `O-HO` specifies that an output at a `HO` name a owns the thread, and that name a must be in the current visibility. In the continuation P (which is not active) the visibility for b is determined by the current visibility $\Delta(*)$ together with b itself, to allow for recursive calls at b. Symmetrically, in rule `I-HO`, a process performing an input at a acquires the thread, with a visibility given by $\Delta(a)$ plus the received name b. A replicated input is well-typed only if it is not active (rule `REP`). When typing parallel composition, we use `split` to insure that at most one of the components holds the thread. We simply remove the name from Δ when typing a restriction (rule `RES-HO`). Visibility plays no role in rules `I-FO` and `O-FO`; the only checks made are given by sequentiality, concerning the respect of the thread.

▶ **Example 7.** Consider

$$P \stackrel{\text{def}}{=} \begin{array}{l} a(x,b).\,\overline{b}\langle x+1\rangle(d).\,a(x',b').\,\overline{b}\langle x+x'\rangle(d').\,\mathbf{0} \\ \mid\ a(x_1,b_1).\,\overline{b_1}\langle x_1\rangle(d_1).\,d_1(d_2).\,\overline{c}\langle x+1\rangle(c_1).\,\mathbf{0} \end{array}$$

The process has three inputs at a; the first two are sequential, and make no requirements on visibility, as their continuations only perform outputs at names received in the input; the third one, in addition, has the effect of creating a name d_1, whose input may activate an output at the free name c. This input requires that in the visibility Δ of P, we find $c \in \Delta(a)$.

▶ **Example 8.** (In this example, we ignore the first-order components of actions, as they are not relevant.) A process of the form

$$P \stackrel{\text{def}}{=} \overline{a}(b).\,(Q \mid c(d).\,\overline{b}(d').\,R)$$

is not typable: the initial visibility of c may not include b, which is created fresh by a; yet, P may perform an output at b after the input at c. Indeed, while P does not directly create a form of higher-order store, it may do so indirectly: the external environment, after completing the treatment of the initial call at a, is supposed to "forget" the higher-order name b. However, this is not the case, as the environment may regain access to b by invoking the (unrelated) name c. In contrast P would become typable (in the straightforward extension of our type system with polyadicity) if the initial output at a would be replaced by $\overline{a}(b,c)$: now c becomes a fresh name, initially unknown, and created in the initial output, together with b, so that the visibility for b and c can be the same and can include both names.

The following lemma shows that a visibility function may always be enlarged (both in the domain and codomain), as long as sequentiality (i.e., being defined on $*$) is respected.

▶ **Lemma 9** (Weakening). *Suppose $\Delta_1 \vdash P$, $\Delta_1 \subseteq \Delta_2$, and ($\Delta_2\downarrow * $ iff $\Delta_1\downarrow *$). Then $\Delta_2 \vdash P$.*

Some other technical lemmas that are used in proofs are presented in Appendix B.1.

▶ **Theorem 10** (Subject reduction). *Suppose $\Delta \vdash P$ and $P \xrightarrow{\mu} P'$. We have:*

- *if μ is a τ action or a first-order input, then $\Delta \vdash P'$; the same holds if μ is a first-order output and $\Delta \not\downarrow *$;*
- *if $\mu = \overline{a}\langle v\rangle(b)$ then $a \in \Delta(*)$ and $\Delta^{-*}; b \mapsto \Delta(*), b \vdash P'$;*
- *if $\mu = a\langle v\rangle(b)$ and $\Delta \not\downarrow *$, then $\Delta; * \mapsto \Delta(a), b \vdash P'$.*

▶ **Example 11.** Transitivity of visibility functions is necessary. Indeed, consider process $P \stackrel{\text{def}}{=} a.\,\overline{b}.\,Q \mid \overline{a}$, and the *non-transitive* function $\Delta = a \mapsto \{a,b\}, * \mapsto a$. We have $\Delta \vdash P$. However $P \xrightarrow{\tau} \overline{b}.\,Q$, with $\Delta \not\vdash \overline{b}.\,Q$, violating Subject Reduction (Theorem 10).

To adapt the generic definitions of behavioural equivalence in Section 2 to the type system for visibility, we only have to specify the meaning of compatibility between typing environments (which in our case are visibility functions). Compatibility means ensuring that processes typed under the two typing environments cannot both be active.

▶ **Definition 12.** Two typings Δ_1 and Δ_2 are *compatible* if $* \notin \mathtt{dom}(\Delta_1) \cap \mathtt{dom}(\Delta_2)$.

We discuss some examples of equivalences in which the behavioural constraints imposed on legal observers by visibility are essential: the equivalences would otherwise be broken.

▶ **Example 13.** Consider a process

$$P \stackrel{\text{def}}{=} \overline{a}(b,c).\,P' \qquad \text{for} \qquad P' \stackrel{\text{def}}{=} (b(d).\,R) \mid c(c_1).\,\overline{a}(b',c').\,c'(c_2).\,\overline{b}(d').\,Q$$

where we assume b does not occur in Q and R. Process P interrogates an external service a, providing access to local services b and c, and the derivative is P'. Now, P' may either be interrogated at b, or at c. In the former case, the output $\overline{b}(d').\,Q$ becomes dead code (as the environment may not install any input at b). In the latter case, the environment is again called at a; now, since visibility ensures us the environment cannot remember b, the input and the output at b in P' may only interact with each other; we can therefore inline the call at b. We can express this behaviour using expansion as follows:

$$P \cong \overline{a}(b,c).\Big(\begin{array}{rl} & b(d).\,(R \mid c(c_1).\,\overline{a}(b',c').\,c'(c_2).\,\mathbf{0}_*) \\ + & c(c_1).\,\overline{a}(b',c').\,c'(c_2).\,\tau.\,\boldsymbol{\nu} d'\,(Q \mid R\{^{d'}/_{d}\})\end{array}\Big)$$

where $\mathbf{0}_*$ indicates an active process that cannot perform any action (e.g., $\boldsymbol{\nu} b\,_2(\overline{b_2}.\,\mathbf{0})$). (This equality holds under any visibility, hence we use the symbol $\cong$.)

If we replaced the second output at a with an output at a different free name, say a', then the processes could still be typable, and the equalities above would still hold. However, the equalities would not hold if we replaced the second output at a with an output at c_1. Indeed, while the environment may not store b or c for using them in a later call at a (or at an unrelated name such as a'), the environment is allowed to use b and c within services that are created following the first call at a; for instance, the environment could be

$$a(b,c).\,\overline{c}(c_1).\,!c_1(c_1').\,\overline{b}(d'').\,\mathbf{0}$$

The following example illustrates the use of visibility to perform garbage-collection on names that will never be used. We refer to Appendix B.3 for the full definition of the corresponding pruning function.

▶ **Example 14.** Consider a typing environment Δ, a Δ-process

$$P \stackrel{\text{def}}{=} !a.\,Q_1 \mid !b.\,\overline{c}.\,!a.\,Q_2 \mid !d.\,\overline{a}.\,\mathbf{0}$$

and a visibility Γ compatible with Δ such that $\Gamma \cup \Delta$ is transitive. Γ is intended here to represent the observer's visibility. Suppose $a \notin \Gamma(*)$; that is, the observer is active but its current visibility does not include a (which may however be in the visibility of other names in Γ). We then have $P \cong^{\Delta}_{\Gamma} !b.\,\overline{c}.\,\mathbf{0}$. The pruning is possible because, intuitively, the transitivity of $\Delta \cup \Gamma$ ensures us that an observer, whenever active, cannot have a in its current visibility. For instance, if the observer interrogates b, then the transitivity of the visibility functions ensures us that $a \notin \Delta(b)$; then $c \in \Delta(b)$ (which is necessary for P to be typable) gives us also $a \notin \Gamma(c)$. Similarly, exploiting transitivity, we deduce that also $d \notin \Gamma(*)$, and that an observer may never be able to interrogate d; thus we can also remove the input on d.

4 Labelled Semantics

4.1 Trace-based proof techniques

The definitions of may-testing and barbed equivalence make use of contextual quantifications. We study here proof techniques for them, using labelled transitions, without such quantifications. These proof techniques will be expressed as forms of trace equivalence and labelled

$$\texttt{TrO-HO}\ \frac{P \xrightarrow{\overline{a}\langle v\rangle(b)} P' \quad a \in \texttt{dom}(\Gamma) \quad \Gamma \not\downarrow *}{[\Gamma]\, P \xrightarrow{\overline{a}\langle v\rangle(b)} [\Gamma; * \mapsto \Gamma(a), b]\, P'} \qquad \texttt{TrI-HO}\ \frac{P \xrightarrow{a\langle v\rangle(b)} P' \quad a \in \Gamma(*)}{[\Gamma]\, P \xrightarrow{a\langle v\rangle(b)} [\Gamma^{-*}; b \mapsto \Gamma(*), b]\, P'}$$

$$\texttt{TrO-FO}\ \frac{P \xrightarrow{\overline{h}\langle v\rangle} P' \quad \Gamma \downarrow *}{[\Gamma]\, P \xrightarrow{\overline{h}\langle v\rangle} [\Gamma]\, P'} \qquad \texttt{TrI-FO}\ \frac{P \xrightarrow{h\langle v\rangle} P' \quad \Gamma \not\downarrow *}{[\Gamma]\, P \xrightarrow{h\langle v\rangle} [\Gamma]\, P'} \qquad \texttt{TrTau}\ \frac{P \xrightarrow{\tau} P' \quad \Gamma \not\downarrow *}{[\Gamma]\, P \xrightarrow{\tau} [\Gamma]\, P'}$$

Figure 2 Visible LTS.

bisimilarity. The definition of may-testing makes use of contextual quantifications. We study here proof techniques without such quantifications, as forms of trace equivalence. For lack of space, analogous proof techniques for barbed equivalence are only reported in [4]. The first step is to express the transitions for a process that are allowed by an observer subject to certain visibility constraints. We call these the *type-allowed transitions*. They are of the form $[\Gamma]\, P \xrightarrow{\mu} [\Gamma']\, P'$, to be read "an observer that has visibility Γ allows process P to make a transition μ and, as a result of the transition, the process becomes P' and the visibility of the observer becomes Γ'". Type-allowed transitions are defined by the rules in Figure 2. Intuitively, an observer typable under Γ can initially test inputs of processes at names in $\Gamma(*)$, if $\Gamma \downarrow *$, and outputs of processes in $\texttt{dom}(\Gamma)$ otherwise. The weak transitions $\Longrightarrow$ and $\xRightarrow{\mu}$, are defined in the usual manner from the strong one; for instance, $[\Gamma]\, P \xRightarrow{\mu} [\Gamma']\, P'$ holds if $[\Gamma]\, P \Longrightarrow [\Gamma]\, P''$, and $[\Gamma]\, P'' \xrightarrow{\mu} [\Gamma']\, P'''$, and $[\Gamma']\, P''' \Longrightarrow [\Gamma']\, P'$, for some P'' and P'''.

An action τ is called a *reduction* (or an *internal action*); the remaining actions are called *interactions*. We use $\mathcal{T}$ to range over *traces*, which are finite sequences of interactions $\mu_1 \dots \mu_n$ $(n \geq 0)$ in which the bound names are all distinct and different from the free names (the definition of bound and free names is as for processes, viewing a trace as a sequence of prefixes). A trace $\mu_1 \dots \mu_n$ is *trace of* P_0 *under visibility* Γ_0 if there are P_i, Γ_i with $[\Gamma_i]\, P_i \xRightarrow{\mu_i} [\Gamma_{i+1}]\, P_{i+1}$, for $0 \leq i < n$.

▶ **Definition 15** (Typed trace equivalence). Two Δ-processes P and Q are *trace equivalent at* Γ, written $P \eqsim^{\Gamma}_{\Delta} Q$, if Δ, Γ are compatible and P, Q have the same sets of traces under Γ.

We show the main compositionality lemmas, and a trace-based characterisation of may testing. Other technical lemmas needed in proofs may be found in Appendix B.2.

▶ **Lemma 16** (Restriction). *If* $P \eqsim^{\Gamma}_{\Delta} Q$ *then also* $\boldsymbol{\nu} a\, P \eqsim^{\Gamma^{-a}}_{\Delta^{-a}} \boldsymbol{\nu} a\, Q$.

▶ **Lemma 17** (Input). *If* $P \eqsim^{\Gamma^{-*}; b \mapsto \Gamma(*), b}_{\Delta; * \mapsto \Delta(a), b} Q$ *and then also* $a(x,b).\, P \eqsim^{\Gamma}_{\Delta} a(x,b).\, Q$.

▶ **Lemma 18** (Output). *If* $P \eqsim^{\Gamma; * \mapsto \Gamma(a), b}_{\Delta^{-*}; b \mapsto \Delta(*), b} Q$ *and then also* $\overline{a}\langle e\rangle(b).\, P \eqsim^{\Gamma}_{\Delta} \overline{a}\langle e\rangle(b).\, Q$.

▶ **Lemma 19** (Parallel composition). *Suppose* $P \eqsim^{\Gamma}_{\Delta} Q$ *and* $(\Gamma_1, \Gamma_2) \in \texttt{split}(\Gamma)$. *Then* $\Gamma_2 \vdash R$ *implies* $P \mid R \eqsim^{\Gamma_1}_{(\Delta \cup \Gamma_2)^+} Q \mid R$ *where* $(-)^+$ *indicates the transitive closure.*

In the proof of Lemma 19, a trace of $P \mid R$ is split into a trace for P and one for R. We have then to ensure that the condition of the lemma are maintained throughout the trace, notwithstanding modifications of the visibility functions. Some care is needed in case of synchronisations between P and R, where the typing of the external observer has to remain the same; we also use the restriction Lemma 16. Other cases also exploit the extension Lemma 36, and the transitivity of the visibility function, to make sure that the views on free names are not increased along the trace.

▶ **Theorem 20** (Characterisation of may testing using trace equivalence). *Suppose* $\Delta \vdash P, Q$ *then* $P \eqsim^{\Gamma}_{\Delta} Q$ *if and only if* $P \cong^{\Gamma}_{\Delta} Q$.

The sketch of the proofs is the same as for characterisations of may testing with trace equivalence in untyped calculi, exploiting substitutivity properties of trace equivalence in the case of Soundness, and reason by contradiction in the case of Completeness. In addition, one has to take care of the observer visibilities, how it changes, and use type-allowed transitions. A similar results of characterisation also hold for barbed equivalence, using a typed form of labelled bisimilarity, and are presented in [4].

We can use typed trace equivalence to prove equalities such as those in Examples 13 and 14; the proofs are simple, following the definition of the equivalence and exploiting type-allowed transitions. See also Appendix B.3 for the proof about the pruning function that generalises Example 14. (Similarly, the proofs can be carried out for barbed equivalence, using the labelled bisimilarity in [4].)

4.2 Visibility property on traces

We formalise the link between our type system and the notion of visibility in game semantics [7]. The reader not familiar with game semantics may safely skip the section. In game semantics, the notions of visibility and well-bracketing are defined based on a justification relation between moves performed by Player. These moves correspond to interactions on higher-order names in our setting, and the notion of justification can be formulated as follows.

▶ **Definition 21.** Let μ_1, μ_2 be two interactions at $\mathtt{HO}$ names. We say that μ_1 *justifies* μ_2, noted $\mu_1 \curvearrowleft \mu_2$, if: (i) either $\mu_1 = a\langle v\rangle(b)$ and $\mu_2 = \overline{b}\langle v'\rangle(c)$, or (ii) $\mu_1 = \overline{a}\langle v\rangle(b)$ and $\mu_2 = b\langle v'\rangle(c)$. When this is not the case, we write $\mu_1 \not\curvearrowleft \mu_2$.

$\mu_1 \curvearrowleft \mu_2$ captures the dependency induced by the fact that μ_2 performs an interaction at a name that has been previously introduced by μ_1. In contrast with the situation in game semantics, though, there is no notion of question and answer in our setting.

We can now define the notion of view. In the definition below, we allow a starting visibility A, because we want to consider processes that already have some knowledge about the observer (Opponent).

▶ **Definition 22** (View). Given $A \subseteq \mathtt{HO}$, we define the *view with starting visibility* A *of a trace* $\mathcal{T}$, written $\mathtt{view}_A(\mathcal{T})$, as follows:

$$\mathtt{view}_A(\varepsilon) = A \qquad \mathtt{view}_A(\mathcal{T}_1\mu_1\mathcal{T}_2\mu_2) = \mathtt{view}_A(\mathcal{T}_1) \cup \mathtt{nHO}(\mu_1) \cup \mathtt{nHO}(\mu_2) \text{ if } \mu_1 \curvearrowleft \mu_2$$

$$\mathtt{view}_A(\mu_0 \dots \mu_n) = A \cup \mathtt{nHO}(\mu_n) \text{ if } \forall 0 \leq i \leq n-1, \mu_i \not\curvearrowleft \mu_n$$

where $\mathtt{nHO}(\mu)$ denotes the set of higher-order names appearing in the action μ.

Views capture the idea that when an action is justified by another one, the names created in between must have been forgotten, otherwise this would mean that the process had the ability to store these names in order to reuse them. In the last clause in Definition 22, the action is justified by an action that happened before the beginning of the trace, therefore it has the starting visibility. In particular, $\mathtt{view}_\emptyset(\mathcal{T})$ yields the usual definition in game semantics.

▶ **Definition 23** (Respecting views). Consider $A \subseteq \mathtt{HO}$, a trace $(\mu_0 \dots \mu_n)$ *respect views under starting visibility* A if $\mathtt{fn}(\mu_0) \subseteq A$ and $\forall 1 \leq i \leq n, \mathtt{fn}(\mu_i) \subseteq \mathtt{view}_A(\mu_0 \dots \mu_{i-1})$

▶ **Lemma 24.** *Consider* Δ, Γ, P *such that* $\Delta \vdash P$, Γ *is compatible with* Δ, *and* $\Delta \cup \Gamma$ *is transitive. Let* $A = (\Gamma \cup \Delta)(*)$, *then every trace of* P *under visibility* Γ *respects views with starting visibility* A.

The idea of the proof is that for a trace $(\mu_0 \dots \mu_n)$ there exist $P_0, \dots, P_{n+1}$, $\Delta_0, \dots, \Delta_n$, $\Gamma_0, \dots, \Gamma_{n+1}$ such that $P_0 = P$, $\Delta_0 = \Delta$, $\Gamma_0 = \Gamma$ and for all $0 \leq i \leq n$, $[\Gamma_i]\, P_i \xrightarrow{\mu_i} [\Gamma_{i+1}]\, P_{i+1}$ and $\Delta_i \vdash P_i$. We show by induction that for each $1 \leq i \leq n$, $(\Gamma_i \cup \Delta_i)(*) \subseteq \mathtt{view}_A(\mu_0 \dots \mu_{i-1})$; it thus follows that the trace respects views under starting visibility A. The condition of $\Delta \cup \Gamma$ being transitive is crucial, otherwise $(\Gamma_i \cup \Delta_i)(*)$ could grow in unwanted ways.

5 Adding Well-Bracketing

5.1 Type system

In the previous sections we have studied visibility in connection with sequentiality. In this section we do the same for a refinement of sequentiality, namely well-bracketing. We begin by reviewing well-bracketing, following [3], and then outline how visibility is accommodated.

From sequentiality to well-bracketing. In languages without control operators, well-bracketing intuitively means that return-call interactions among terms follow a stack-based discipline. In a well-bracketed system, a service that is interrogated, say S, acquires the thread and is supposed to return a final result (unless the computation diverges) thus releasing the thread. During its computation, S may however interrogate another service, say T, which, upon completion of its computation, will return the result to S; service S must receive the answer from T before returning the final result. The implementation of this policy requires *continuation names*, that are used to handle the return messages. When a service is interrogated, it receives a fresh continuation name that is used once, in output, to deliver the final result. Accordingly, a client that has interrogated the service will use the continuation name in input to receive the result. Thus continuation names are *linear* [11] – they may only be used once. In contrast with [3], we do not require *receptiveness* [23] – the input-end of the name needs not be made available as soon as the name is created. Enforcing only linearity allows us to gain flexibility. To ensure the well-bracketing policy, and linearity of continuation names, the type system in [20] makes use of stacks.

▶ **Definition 25** ([20])**.** A stack, σ, is a sequence of input- and output- tagged continuation names, in which the input and output tags alternate:

$$\sigma ::= \sigma_{\mathtt{O}} \mid \sigma_{\mathtt{I}} \qquad \sigma_{\mathtt{O}} ::= p^{\mathtt{O}}, \sigma_{\mathtt{I}} \mid \emptyset \qquad \sigma_{\mathtt{I}} ::= p^{\mathtt{I}}, \sigma_{\mathtt{O}} \mid \emptyset$$

Moreover: a name may appear at most once with a given tag; and, if a name appears with both tags, then the input occurrence should immediately follow the output occurrence.

We write $p^{\mathtt{O}} \in \sigma$ (resp. $p^{\mathtt{I}} \in \sigma$) if there is an occurrence of $p^{\mathtt{O}}$ (resp. $p^{\mathtt{I}}$) in σ. We write $p \in \sigma$ if either $p^{\mathtt{O}} \in \sigma$ or $p^{\mathtt{I}} \in \sigma$.

Intuitively, a stack expresses the expected usage of the free continuation names in a process. For instance, if P can be typed using stack $p_1 : \mathtt{o}, p_2 : \mathtt{i}, p_3 : \mathtt{o}, p_3 : \mathtt{i}, p_4 : \mathtt{o}$, then $p_1, .., p_4$ are the free continuation names in P; among these, p_1 will be used first, in an output; then p_2 will be used, in an input interaction with the environment. P possesses both the output and the input capability on p_3, so it shall perform a reduction at p_3; the computation for P terminates with an output at p_4. This behaviour however concerns only the free continuation names of P: at any time when an output usage is expected, P may decide to create a new continuation name and send it out, maintaining its input end.

$$\texttt{O-ser}\ \frac{a \in A \quad \langle \Delta; b \mapsto A, b; p \mapsto A, b \mid p^{\mathrm{I}}, \sigma\rangle \vdash P}{\langle \Delta; * \mapsto A \mid \sigma\rangle \vdash \overline{a}\langle e\rangle(b,p).\, P} \qquad \texttt{O-con}\ \frac{p \in A \quad \langle \Delta; b \mapsto A, b \mid \sigma\rangle \vdash P}{\langle \Delta; * \mapsto A \mid p^{\mathrm{O}}, \sigma\rangle \vdash \overline{p}\langle e\rangle(b).\, P}$$

$$\texttt{I-ser}_1\ \frac{\langle \Delta; a \mapsto A; * \mapsto A, b, p \mid p^{\mathrm{O}}, \sigma\rangle \vdash P}{\langle \Delta; a \mapsto A \mid \sigma\rangle \vdash a(x,b,p).\, P} \qquad \texttt{I-ser}_2\ \frac{\langle \Delta; a \mapsto A; * \mapsto A, b, p \mid p^{\mathrm{O}}\rangle \vdash P}{\langle \Delta; a \mapsto A \mid \emptyset\rangle \vdash\, !a(x,b,p).\, P}$$

$$\texttt{I-con}\ \frac{\langle \Delta; * \mapsto A, b \mid q^{\mathrm{O}}\rangle \vdash P}{\langle \Delta; p \mapsto A \mid p^{\mathrm{I}}, q^{\mathrm{O}}\rangle \vdash p(x,b).\, P} \qquad \texttt{RES-ser}\ \frac{\langle \Delta \mid \sigma\rangle \vdash P}{\langle \Delta^{-a} \mid \sigma\rangle \vdash \nu a P}$$

$$\texttt{RES-con}_1\ \frac{\langle \Delta \mid \sigma_1, p^{\mathrm{O}}, p^{\mathrm{I}}, \sigma_2\rangle \vdash P}{\langle \Delta^{-p} \mid \sigma_1, \sigma_2\rangle \vdash \nu p P} \qquad \texttt{RES-con}_2\ \frac{\langle \Delta \mid \sigma\rangle \vdash P \quad p \notin \sigma}{\langle \Delta^{-p} \mid \sigma\rangle \vdash \nu p P}$$

$$\texttt{PAR}\ \frac{\begin{array}{c}\langle \Delta_1 \mid \sigma_1\rangle \vdash P_1 \quad (\Delta_1, \Delta_2) \in \texttt{split}(\Delta) \\ \langle \Delta_2 \mid \sigma_2\rangle \vdash P_2 \quad \sigma \in \texttt{inter}(\sigma_1, \sigma_2)\end{array}}{\langle \Delta \mid \sigma\rangle \vdash P_1 \mid P_2} \qquad \texttt{NIL}\ \frac{\Delta \not/ *}{\langle \Delta \mid \emptyset\rangle \vdash \mathbf{0}}$$

Figure 3 Typing rules with visibility and stack.

Visibility. We now discuss how to combine the visibility in Section 3 with well-bracketing. Higher-order names are partitioned into the set SER of *server names*, and the set CON of *continuation names*. Now a, b, c range over server names, while p, q, r range over continuation names. By convention, server names carry a triple consisting of a first-order value, a server name and a continuation; whereas continuations carry a first-order value and a server name. Accordingly, outputs on higher-order names are of the form $\overline{a}\langle n\rangle(b,p)$ and $\overline{p}\langle n\rangle(b)$, and similarly for inputs. First-order names are as before.

Visibility functions are then transitive partial functions from $\texttt{SER} \cup \texttt{CON} \cup \{*\}$ to $\mathcal{P}_{\texttt{fin}}(\texttt{SER} \cup \texttt{CON})$. These functions handle server and continuation names essentially like HO names in Section 3.

Typing environments include a stack; thus a typing environment is a pair $\langle \Delta \mid \sigma\rangle$ of a visibility function and a stack. In the typing rule for parallel composition, we rely on splitting (defined as in Section 3) as well as on $\texttt{inter}(\sigma_1, \sigma_2)$, the interleaving of stacks σ_1 and σ_2.

▶ **Definition 26** (Interleaving). We write $\sigma_1 \in \texttt{inter}(\sigma_2; \sigma_3)$ if (i) σ_1 is a stack, and (ii) σ is an interleaving of σ_2 and σ_3 as by the following inductive rules:

1. $\emptyset \in \texttt{inter}(\emptyset; \emptyset)$
2. $p : \texttt{o}, \sigma_1 \in \texttt{inter}(\sigma_2; \texttt{p} : \texttt{o}, \sigma_3)$ if $\sigma_1 \in \texttt{inter}(\sigma_2; \sigma_3)$
3. $p : \texttt{o}, \sigma_1 \in \texttt{inter}(\texttt{p} : \texttt{o}, \sigma_2; \sigma_3)$ if $\sigma_1 \in \texttt{inter}(\sigma_2; \sigma_3)$
4. the same as (2) and (3) with $p : \texttt{i}$ instead of $p : \texttt{o}$

If a name appears both in σ_2 and in σ_3 with the same tag, then $\texttt{inter}(\sigma_2; \sigma_3)$ is empty. Figure 3 gives the typing rules. The rules combine the handling of visibility and the update of the stack. For instance, in rule O-ser, newly created names b and p are added to the visibility when typing the continuation P, and p^{I} is added on top of the stack, meaning that P shall perform an input at p before interacting on other continuation names. The output capability on p is transmitted along the output at a; correspondingly, in rules $\texttt{I-ser}_1$ and $\texttt{I-ser}_2$, p^{O} is added to σ. In contrast with [3], we do not impose $\sigma = \emptyset$ in rule $\texttt{I-ser}_1$, as we do not enforce input receptiveness on continuation names. Performing inputs and outputs at continuation names has the effect of consuming the corresponding particle on top of the stack. The stack is empty when typing a replicated input, so to avoid duplication of continuation names. Rules SUCC, O-FO, I-FO, RES-FO, SUM and IF are like in Figure 1, and are omitted.

▶ **Theorem 27** (Subject reduction). *Suppose* $\langle \Delta \mid \sigma \rangle \vdash P$ *and* $P \xrightarrow{\mu} P'$. *We have:*

- *if* $\mu = \tau$, *then either* $\sigma = p^{\mathcal{O}}, p^{\mathcal{I}}, \sigma''$ *with* $p \in \mathtt{Ncon} \setminus \mathtt{fn}(P')$, *in which case* $\langle \Delta^{-p} \mid \sigma'' \rangle \vdash P'$, *or* $\langle \Delta \mid \sigma \rangle \vdash P'$;
- *if* $\mu = \overline{a}\langle v \rangle(b, p)$ *then* $a \in \Delta(*)$ *and* $\langle \Delta^{-*}; b \mapsto \Delta(*), b; p \mapsto \Delta(*), b \mid p^{\mathcal{I}}, \sigma \rangle \vdash P'$;
- *if* $\mu = \overline{p}\langle v \rangle(b)$ *then* $\sigma = p^{\mathcal{O}}, \sigma''$, $p \in \Delta(*)$ *and* $\langle \Delta^{-*}; b \mapsto \Delta(*), b \mid \sigma'' \rangle \vdash P'$;
- *if* $\mu = a\langle v \rangle(b, p)$ *and* $\Delta \not\downarrow *$ *then* $\langle \Delta; * \mapsto \Delta(a), b, p \mid p^{\mathcal{O}}, \sigma \rangle \vdash P'$;
- *if* $\mu = p\langle v \rangle(b)$ *and* $\Delta \not\downarrow *$ *then* $\sigma = p^{\mathcal{I}}, q^{\mathcal{O}}, \sigma''$ *and* $\langle \Delta^{-p}; * \mapsto \Delta(p), b \mid q^{\mathcal{O}}, \sigma'' \rangle \vdash P'$;
- *if* $\mu = h\langle v \rangle$ *or* $(\mu = \overline{h}\langle v \rangle$ *and* $\Delta \not\downarrow *)$ *then* $\langle \Delta \mid \sigma \rangle \vdash P'$.

As previously, we have to define compatibility of typing environments so to be able to use the contextual forms of behavioural equivalence from Section 2.

▶ **Definition 28.** $\langle \Delta_1 \mid \sigma_1 \rangle$ and $\langle \Delta_2 \mid \sigma_2 \rangle$ are compatible if Δ_1 and Δ_2 are compatible (as by Definition 12) and $\mathtt{inter}(\sigma_1, \sigma_2) \neq \emptyset$.

▶ **Example 29.** Consider the process

$$\begin{aligned} P \stackrel{\text{def}}{=} \boldsymbol{\nu} h \,(\ & \overline{h}\langle 5 \rangle \mid \\ & \overline{a}(b, q).\,(\ !b(p).\, h(x).\, (\overline{h}\langle x+1 \rangle.\, \mathbf{0} \mid \overline{p}.\, \mathbf{0}) \mid \\ & \qquad q.\, h(x).\, \overline{h}\langle x \rangle.\, \overline{c}(d, r).\, r.\, h(y).\, \overline{h}\langle y \rangle.\, \mathtt{if}\ x = y\ \mathtt{then}\ \overline{s}.\, \mathbf{0}\ \mathtt{else}\ \mathbf{0}_*)) \end{aligned}$$

Here h is used as an integer reference, initially set to 5. The process calls a service a, giving access to a name b that can be used to increment the reference. When the call at a returns (at q), the reference is read twice and, in between, a call at c is made. Without visibility, the two values read for h might be different: as b has been exported, the environment could freely use b and increment the reference in between the two reads. Visibility ensures us that, when the initial call at a is returned, name b is "forgotten" by the environment, which cannot therefore use it anymore. Thus, if Q is the process defined like P but with the last line replaced with $q.\, \overline{c}(d, r).\, r.\, \overline{s}.\, \mathbf{0}$, then we have $P \cong Q$. We refer to Appendix C.1 for an ML-like presentation of the example above, as well as for a longer example, also showing how re-entrant calls may sometimes make visibility boundaries hard to predict.

5.2 Labelled semantics

To represent the point of view of the observer, we use typing environments of the form $[\Gamma \mid \kappa]$, where Γ is a visibility function and κ is a *stack*. Figure 4 defines the typed transitions, written $[\Gamma \mid \kappa]P \xrightarrow{\mu} [\Gamma' \mid \kappa']P'$. Rules `TrTau`, `TrO-fo` and `TrI-fo` are similar to their counterpart in Figure 2, and are omitted.

A trace $\mu_1 \dots \mu_n$ is a *well-bracketed trace of* P_0 *under* $[\Gamma_0 \mid \kappa_0]$ if there are P_i, Γ_i, κ_i $(1 \leq i \leq n)$ with $[\Gamma_i \mid \kappa_i]P_i \stackrel{\mu_i}{\Longrightarrow} [\Gamma_{i+1} \mid \kappa_{i+1}]P_{i+1}$ for all $0 \leq i < n$.

▶ **Definition 30** (Typed trace equivalence). Two $\langle \Delta \mid \sigma \rangle$-processes P and Q are *trace equivalent at* $[\Gamma \mid \kappa]$, written $P \eqsim^{[\Gamma \mid \kappa]}_{\langle \Delta \mid \sigma \rangle} Q$, if $\langle \Delta \mid \sigma \rangle$ and $[\Gamma \mid \kappa]$ are compatible and P and Q have the same sets of well-bracketed traces under $[\Gamma \mid \kappa]$.

This typed version of trace equivalence enjoys properties, including substitutivity properties, similar to those of the trace equivalence in Section 4. We only report the characterisation result with respect to may testing.

▶ **Theorem 31** (Characterisation of may testing using trace equivalence). *Suppose* $\langle \Delta \mid \sigma \rangle \vdash P, Q$; *then* $P \eqsim^{[\Gamma \mid \kappa]}_{\langle \Delta \mid \sigma \rangle} Q$ *if and only if* $P \cong^{[\Gamma \mid \kappa]}_{\langle \Delta \mid \sigma \rangle} Q$.

The analogous result for typed bisimilarity is in Appendix C, together with results about the shape of type-allowed traces akin to those in Section 4.2.

$$\texttt{TrO-ser}\ \frac{P \xrightarrow{\overline{a}\langle v\rangle(b,p)} P' \quad a \in \texttt{dom}(\Gamma) \quad \Gamma \not\downarrow *}{[\Gamma \mid \kappa]P \xrightarrow{\overline{a}\langle v\rangle(b,p)} [\Gamma; * \mapsto \Gamma(a), b, p \mid p^{\texttt{O}}, \kappa]P'}$$

$$\texttt{TrO-con}\ \frac{P \xrightarrow{\overline{p}\langle v\rangle(b)} P' \quad p \in \texttt{dom}(\Gamma) \quad \Gamma \not\downarrow *}{[\Gamma \mid p^{\texttt{I}}, q^{\texttt{O}}, \kappa]P \xrightarrow{\overline{p}\langle v\rangle(b)} [\Gamma^{-p}; * \mapsto \Gamma(p), b \mid q^{\texttt{O}}, \kappa]P'}$$

$$\texttt{TrI-ser}\ \frac{P \xrightarrow{a\langle v\rangle(b,p)} P' \quad a \in \Gamma(*)}{[\Gamma \mid \kappa]P \xrightarrow{a\langle v\rangle(b,p)} [\Gamma^{-*}; p \mapsto \Gamma(*), b; b \mapsto \Gamma(*), b \mid p^{\texttt{I}}, \kappa]P'}$$

$$\texttt{TrI-con}\ \frac{P \xrightarrow{p\langle v\rangle(b)} P' \quad p \in \Gamma(*)}{[\Gamma \mid p^{\texttt{O}}, \kappa]P \xrightarrow{p\langle v\rangle(b)} [\Gamma^{-*}; b \mapsto \Gamma(*), b \mid \kappa]P'}$$

Figure 4 Visible LTS with stack.

6 Conclusions and Future Work

In this paper we have studied the meaning of visibility, originating from game-semantics interpretations of typed λ-calculi and reflecting absence of higher-order store, in a calculus of pure name-passing such as the π-calculus. In contrast with λ-calculi, in the π-calculus there is no explicit notion of store, and no unique identities (e.g., the input end of a name may have multiple occurrences, arbitrarily structured). These aspects determine considerable differences in the semantic analysis of visibility. For instance, in the π-calculus visibility has to be enforced by means of a dedicated type system; and the visibility functions have to satisfy special properties such as transitivity. We have highlighted such differences, and investigated the behavioural effects of visibility. In particular, we have shown full abstract characterisations of may testing and barbed equivalence, as forms of trace equivalence and labelled bisimilarity.

There are several directions for future work that we would like to pursue. In game semantics, further aspects related to first-order store have been studied; for instance allowing first-order references as values (that can be passed around) [13, 9], or allowing to store first-order references (i.e., references with types such as `int ref ref` [17, 10]). The distinction between first-order and higher-order state has also been studied in [14]. We would like to investigate the effects of such distinctions on our types and proof techniques.

We have worked with πI, a dialect of π-calculus in which all names exchanged are newly created. Thus πI lacks the free-output construct, where the communication of existing names is permitted. As shown in the literature [5, 25, 8], πI is closer to game semantics than the standard π-calculus. We would like to see whether our results can be adapted so to accommodate also the free output construct. While free output can be modelled in πI [2], it allows a direct representation of idioms such as "delegate to someone else the task of answering', or the expression of tail-call recursion.

We have studied visibility on top of type systems for sequentiality and well-bracketing. Allowing concurrency would probably require relaxing the definition of the visibility functions; we leave the details to future work.

References

1 Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco T. Vasconcelos, and Nobuko Yoshida. Behavioral types in programming languages. *Found. Trends Program. Lang.*, 3(2-3):95–230, 2016. doi:10.1561/2500000031.

2 Michele Boreale. On the expressiveness of internal mobility in name-passing calculi. *Theor. Comput. Sci.*, 195(2):205–226, 1998. doi:10.1016/S0304-3975(97)00220-X.

3 Daniel Hirschkoff, Enguerrand Prebet, and Davide Sangiorgi. On sequentiality and well-bracketing in the π-calculus. In *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. IEEE, June 2021. doi:10.1109/lics52264.2021.9470559.

4 Daniel Hirschkoff, Iwan Quémerais, and Davide Sangiorgi. First-order store and visibility in name-passing calculi. *CoRR*, abs/2504.17350, 2025. Extended online version of this paper. doi:10.48550/arXiv.2504.17350.

5 Kohei Honda. Processes and games. In Fabio Gadducci and Ugo Montanari, editors, *Fourth International Workshop on Rewriting logic and Its Applications, WRLA2002, Pisa, Italy, 19-21, 2002*, volume 71 of *Electronic Notes in Theoretical Computer Science*, pages 40–69. Elsevier, 2002. doi:10.1016/S1571-0661(05)82528-9.

6 Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In Chris Hankin, editor, *Programming Languages and Systems - ESOP'98, 7th European Symposium on Programming, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 1998. doi:10.1007/BFB0053567.

7 Guilhem Jaber and Andrzej S. Murawski. Complete trace models of state and control. In *Programming Languages and Systems: 30th European Symposium on Programming, ESOP 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 – April 1, 2021, Proceedings*, pages 348–374, Berlin, Heidelberg, 2021. Springer-Verlag. doi:10.1007/978-3-030-72019-3_13.

8 Guilhem Jaber and Davide Sangiorgi. Games, Mobile Processes, and functions. In *CSL 2022 - 30th EACSL Annual Conference on Computer Science Logic*, pages 1–35, Göttingen, Germany, February 2022. URL: https://hal.science/hal-03407123.

9 Guilhem Jaber and Nikos Tzevelekos. Trace semantics for polymorphic references. In Martin Grohe, Eric Koskinen, and Natarajan Shankar, editors, *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016*, pages 585–594. ACM, 2016. doi:10.1145/2933575.2934509.

10 Ohad Kammar, Paul B. Levy, Sean K. Moss, and Sam Staton. A monad for full ground reference cells. In *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, Reykjavik, Iceland, 2017. IEEE. doi:10.1109/LICS.2017.8005109.

11 Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the pi-calculus. *ACM Trans. Program. Lang. Syst.*, 21(5):914–947, 1999. doi:10.1145/330249.330251.

12 Naoki Kobayashi and Davide Sangiorgi. A hybrid type system for lock-freedom of mobile processes. *ACM Trans. Program. Lang. Syst.*, 32(5):16:1–16:49, 2010. doi:10.1145/1745312.1745313.

13 James Laird. A fully abstract trace semantics for general references. In Lars Arge, Christian Cachin, Tomasz Jurdzinski, and Andrzej Tarlecki, editors, *Automata, Languages and Programming, 34th International Colloquium, ICALP 2007, Wroclaw, Poland, July 9-13, 2007, Proceedings*, volume 4596 of *Lecture Notes in Computer Science*, pages 667–679. Springer, 2007. doi:10.1007/978-3-540-73420-8_58.

14 Paul-André Melliès and Nicolas Tabareau. An algebraic account of references in game semantics. In Samson Abramsky, Michael W. Mislove, and Catuscia Palamidessi, editors, *Proceedings of*

the 25th Conference on Mathematical Foundations of Programming Semantics, MFPS 2009, Oxford, UK, April 3-7, 2009, volume 249 of *Electronic Notes in Theoretical Computer Science*, pages 377–405. Elsevier, 2009. doi:10.1016/J.ENTCS.2009.07.099.

15 R. Milner. The polyadic π-calculus: a tutorial. Technical Report ECS–LFCS–91–180, LFCS, 1991. Also in *Logic and Algebra of Specification*, ed. F.L. Bauer, W. Brauer and H. Schwichtenberg, Springer Verlag, 1993.

16 Robin Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2(2):119–141, June 1992. doi:10.1017/S0960129500001407.

17 Andrzej S. Murawski and Nikos Tzevelekos. Algorithmic games for full ground references. *Formal Methods in System Design*, 52(3):277–314, 2018. doi:10.1007/s10703-017-0292-9.

18 Luca Padovani. Deadlock and lock freedom in the linear π-calculus. In Thomas A. Henzinger and Dale Miller, editors, *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014*, pages 72:1–72:10. ACM, 2014. doi:10.1145/2603088.2603116.

19 Benjamin C. Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. *Math. Struct. Comput. Sci.*, 6(5):409–453, 1996. doi:10.1017/S096012950007002X.

20 Enguerrand Prebet. *Typed Behavioural Equivalences in the Pi-Calculus*. Theses, Ecole normale supérieure de lyon - ENS LYON ; Università degli studi (Bologne, Italie), September 2022. URL: https://hal.science/tel-03920089.

21 Davide Sangiorgi. The lazy lambda calculus in a concurrency scenario. *Inf. Comput.*, 111(1):120–153, 1994. doi:10.1006/INCO.1994.1042.

22 Davide Sangiorgi. pi-calculus, internal mobility, and agent-passing calculi. *Theor. Comput. Sci.*, 167(1&2):235–274, 1996. doi:10.1016/0304-3975(96)00075-8.

23 Davide Sangiorgi. The name discipline of uniform receptiveness. *Theor. Comput. Sci.*, 221(1-2):457–493, 1999. doi:10.1016/S0304-3975(99)00040-7.

24 Nobuko Yoshida, Martin Berger, and Kohei Honda. Strong normalisation in the pi -calculus. *Inf. Comput.*, 191(2):145–202, 2004. doi:10.1016/J.IC.2003.08.004.

25 Nobuko Yoshida, Simon Castellan, and Léo Stefanesco. Game semantics: Easy as pi. *CoRR*, abs/2011.05248, 2020. arXiv:2011.05248.

A Section 2: Definition of the Operational Semantics

Structural congruence, written $\equiv$, is the smallest relation that is an equivalence relation, contains α-conversion, is preserved by all operators of the calculus, and satisfies the following axioms:

$$\frac{}{P \mid Q \;\equiv\; Q \mid P} \qquad \frac{}{P \mid (Q \mid R) \;\equiv\; (P \mid Q) \mid R} \qquad \frac{}{P \mid 0 \equiv P} \qquad \frac{}{P + Q \;\equiv\; Q + P}$$

$$\frac{}{P + (Q + R) \;\equiv\; (P + Q) + R} \qquad \frac{a \notin \mathtt{fn}(Q)}{(\boldsymbol{\nu} aP) \mid Q \;\equiv\; \boldsymbol{\nu} a(P \mid Q)} \qquad \frac{}{\boldsymbol{\nu} a \boldsymbol{\nu} b P \equiv \boldsymbol{\nu} b \boldsymbol{\nu} a P}$$

$$\frac{}{!\alpha.\, P \equiv !\alpha.\, P \mid \alpha.\, P}$$

Similar versions of axioms for restrictions of the form $\boldsymbol{\nu} h$ instead of $\boldsymbol{\nu} a$ have been omitted.

To define the labelled transition system (LTS), we introduce *actions*, defined by the following grammar:

$$\mu \quad ::= \quad \tau \;\mid\; a\langle v\rangle(b) \;\mid\; \overline{a}\langle v\rangle(b) \;\mid\; h\langle v\rangle \;\mid\; \overline{h}\langle v\rangle \;\mid\; \omega$$

Actions are *early* for first-order values, and *late* for the transmission of `HO` names. We write $\mathtt{bn}(\mu)$ (resp. $\mathtt{fn}(\mu)$) for the bound (resp. free) names of action μ, defined by saying that b is a bound name in actions $a\langle v\rangle(b)$ and $\overline{a}\langle v\rangle(b)$, and the other names in actions are free.

$$\frac{}{a(x,b).\,P \xrightarrow{a\langle v\rangle(b)} P[v/x]} \qquad \frac{e \Downarrow v}{\overline{a}\langle e\rangle(b).\,P \xrightarrow{\overline{a}\langle v\rangle(b)} P} \qquad \frac{}{h(x).\,P \xrightarrow{h\langle v\rangle} P[v/x]}$$

$$\frac{e \Downarrow v}{\overline{h}\langle e\rangle.\,P \xrightarrow{\overline{h}\langle v\rangle} P} \qquad \frac{}{\omega.\,P \xrightarrow{\omega} \mathbf{0}} \qquad \frac{\alpha.\,P \xrightarrow{\mu} Q}{!\alpha.\,P \xrightarrow{\mu} Q \mid !\alpha.\,P} \qquad \frac{P \xrightarrow{\mu} Q}{\boldsymbol{\nu} a\ P \xrightarrow{\mu} \boldsymbol{\nu} a\ Q}\, a \notin \mathtt{fn}(\mu)$$

$$\frac{P \xrightarrow{\mu} Q}{\boldsymbol{\nu} h\ P \xrightarrow{\mu} \boldsymbol{\nu} h\ Q}\, h \notin \mathtt{fn}(\mu) \qquad \frac{P \xrightarrow{\mu} P'}{P + Q \xrightarrow{\mu} P'} \qquad \frac{P \xrightarrow{a\langle v\rangle(b)} P' \quad Q \xrightarrow{\overline{a}\langle v\rangle(b)} Q'}{P \mid Q \xrightarrow{\tau} \boldsymbol{\nu} b(P' \mid Q')}$$

$$\frac{P \xrightarrow{h\langle v\rangle} P' \quad Q \xrightarrow{\overline{h}\langle v\rangle} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'} \qquad \frac{P \xrightarrow{\mu} P'}{P \mid Q \xrightarrow{\mu} P' \mid Q}\, \mathtt{fn}(Q) \cap \mathtt{bn}(\mu) = \emptyset$$

$$\frac{e \Downarrow v \quad f \Downarrow v \quad P \xrightarrow{\mu} P'}{\mathtt{if}\ e = f\ \mathtt{then}\ P\ \mathtt{else}\ Q \xrightarrow{\mu} P'} \qquad \frac{e \Downarrow v \quad f \Downarrow v' \quad v \neq v' \quad Q \xrightarrow{\mu} Q'}{\mathtt{if}\ e = f\ \mathtt{then}\ P\ \mathtt{else}\ Q \xrightarrow{\mu} Q'}$$

Figure 5 Labelled transition semantics.
Symmetric versions of the rules for parallel composition and for sum have been omitted.

We suppose that any closed first-order expression e can be evaluated to some value v. We write $e \Downarrow v$ for the corresponding evaluation relation. We furthermore suppose that equality between values can be tested.

The LTS is defined, on closed processes, by the rules in Figure 5. Based on $\xrightarrow{\mu}$, we define the following relations: $\longrightarrow$ stands for $\xrightarrow{\tau}$, and $\Longrightarrow$ stands for the reflexive transitive closure of $\longrightarrow$. We write $\overset{\mu}{\Longrightarrow}$ for $\Longrightarrow\xrightarrow{\mu}\Longrightarrow$, and $\overset{\hat{\mu}}{\Longrightarrow}$ is equal to $\overset{\mu}{\Longrightarrow}$ when $\mu \neq \tau$, while $\overset{\hat{\tau}}{\Longrightarrow} = \Longrightarrow$.

We write $P \downarrow_\omega$ if $P \xrightarrow{\omega}$, i.e., if P contains an unguarded occurrence of an ω prefix at toplevel. We write $P \not\downarrow_\omega$ when this is not the case.

B Additional Material for Section 3

B.1 Properties of the type system

▶ **Lemma 32.** *Suppose $\Delta \vdash P$ and $a \notin \mathtt{fn}(P)$. Then also $\Delta^{-a} \vdash P$.*

▶ **Lemma 33** (Invariance of typing w.r.t. structural congruence)**.** *If $\langle \Delta \mid \sigma\rangle \vdash P$ and $P \equiv P'$, then also $\langle \Delta \mid \sigma\rangle \vdash P'$.*

B.2 Properties of trace equivalence

▶ **Lemma 34** (Shrinking)**.** *Suppose $\Gamma' \subseteq \Gamma$, Δ and Γ' compatible and $P \mathrel{\dot{\simeq}}^{\Gamma}_{\Delta} Q$. Then also $P \mathrel{\dot{\simeq}}^{\Gamma'}_{\Delta} Q$.*

▶ **Definition 35** (Projection)**.** Let Γ be a visibility function and N a set of names. We define a visibility function as follows:

$$\mathtt{proj}(N,\Gamma) : \begin{array}{rcl} \mathtt{HO} \cup \{*\} & \to & \mathcal{P}(\mathtt{HO}) \\ a \in \mathrm{dom}(\Gamma) \cap N & \mapsto & \Gamma(a) \cap N \end{array}$$

▶ **Lemma 36** (Extension). *Consider* $\Delta, \Gamma, \Gamma', P, Q$ *such that* $\mathtt{proj}(\mathtt{fn}(\Gamma, P, Q), \Gamma') = \Gamma$ *and* Δ *and* Γ' *are compatible. If* $P \eqsim^{\Gamma}_{\Delta} Q$ *then* $P \eqsim^{\Gamma'}_{\Delta} Q$.

In the statement above, $\mathtt{fn}(\Gamma, P, Q)$ stands for the union of all names mentioned in Γ, together with the names occurring free in P and Q.

B.3 Pruning Processes using Types

Consider a visibility function Δ and $S \subseteq \mathtt{HO}$. We write $\alpha \frown S$ when α is an input of the form $a(x, b)$ and either $a \in S$ or ($a \in \mathtt{dom}(\Delta)$ and $\Delta(a) \cap S \neq \emptyset$).

We define inductively the pruning function on processes that removes all instances of the names in S as follows:

$$\mathtt{Pr}^{\Delta}_{S}(\alpha.\, P) = \begin{cases} \mathbf{0} \text{ if } \alpha \frown S \\ \alpha.\, \mathtt{Pr}^{\Delta}_{S}(P) \text{ otherwise} \end{cases} \qquad \mathtt{Pr}^{\Delta}_{S}(!\alpha.\, P) = \begin{cases} \mathbf{0} \text{ if } \alpha \frown S \\ !\alpha.\, \mathtt{Pr}^{\Delta}_{S}(P) \text{ otherwise} \end{cases}$$

$$\mathtt{Pr}^{\Delta}_{S}(\overline{\alpha}.\, P) = \overline{\alpha}.\, \mathtt{Pr}^{\Delta}_{S}(P) \qquad \mathtt{Pr}^{\Delta}_{S}(\boldsymbol{\nu} a\ P) = \boldsymbol{\nu} a\ \mathtt{Pr}^{\Delta}_{S}(P) \qquad \mathtt{Pr}^{\Delta}_{S}(\boldsymbol{\nu} h\ P) = \boldsymbol{\nu} h\ \mathtt{Pr}^{\Delta}_{S}(P)$$

$$\mathtt{Pr}^{\Delta}_{S}(P \mid Q) = \mathtt{Pr}^{\Delta}_{S}(P) \mid \mathtt{Pr}^{\Delta}_{S}(Q) \qquad \mathtt{Pr}^{\Delta}_{S}(P + Q) = \mathtt{Pr}^{\Delta}_{S}(P) + \mathtt{Pr}^{\Delta}_{S}(Q) \qquad \mathtt{Pr}^{\Delta}_{S}(\mathbf{0}) = \mathbf{0}$$

$$\mathtt{Pr}^{\Delta}_{S}(\omega) = \omega \qquad \mathtt{Pr}^{\Delta}_{S}(\mathtt{if}\ e = f\ \mathtt{then}\ P\ \mathtt{else}\ Q) = \mathtt{if}\ e = f\ \mathtt{then}\ \mathtt{Pr}^{\Delta}_{S}(P)\ \mathtt{else}\ \mathtt{Pr}^{\Delta}_{S}(Q)$$

▶ **Lemma 37** (Pruning). *Let* Γ, Δ, P *such* $\Delta \vdash P$ *and* Γ *and* Δ *are compatible and* $\Delta \cup \Gamma$ *is transitive. Then for all* $S \subseteq \mathtt{HO}$ *such that* $S \cap (\Gamma \cup \Delta(*)) = \emptyset$, $\Delta^{-S} \vdash \mathtt{Pr}^{\Delta}_{S}(P)$ *and* $P \eqsim^{\Gamma}_{\Delta} \mathtt{Pr}^{\Delta}_{S}(P)$.

We actually remove more than the names in S. Indeed, to preserve sequentiality, we cannot simply remove an output; we actually remove all inputs that may cause an output on the names in S.

The idea of the proof is to show that $S \cap (\Gamma \cup \Delta(*)) = \emptyset$ is an invariant throughout the trace; for this we use repeatedly transitivity of $\Gamma \cup \Delta$.

Proof. Let $\mu_1 \dots \mu_n$ a trace of P under Γ, let P_i, Γ_i be the processes and visibility functions given that trace and Δ_i the visibility functions given by subject reduction. We show by induction on i that there exists Γ'_i, Δ'_i such that $\Gamma'_i \supseteq \Gamma_i$ and $\Delta'_i \supseteq \Delta_i$ and $S \cap (\Gamma'_i \cup \Delta'_i)(*) = \emptyset$ and that Δ'_i and Γ'_i are compatible, have a transitive union and coincide on the intersection of their domains. The case $i = 0$ is immediate, let $0 \leq i < n$, we assume the induction hypothesis on i and show it for $i + 1$ by cases on μ_i :

- if $\mu_i = \tau$, or $\mu_i = h\langle v \rangle$ or $\mu_i = \overline{h}\langle v \rangle$ then $\Gamma_{i+1} = \Gamma_i$ and $\Delta_{i+1} = \Delta_i$, these cases are immediate by setting $\Gamma'_{i+1} = \Gamma'_i$ and $\Delta'_{i+1} = \Delta'_i$.
- if $\mu_i = \overline{a}\langle v \rangle(b)$ then $\Gamma_{i+1} = \Gamma_i; * \mapsto \Gamma_i(a), b$ and $\Delta_{i+1} = \Delta_i^{-*}; b \mapsto \Delta_i(*), b$. Let $\Gamma'_{i+1} = \Gamma'_i; * \mapsto \Delta'_i(*), b \supseteq \Gamma_{i+1}$ because $a \in \Delta_i(*)$ therefore $\Gamma_i(a) \subseteq \Delta_i(*)$. And let $\Delta'_{i+1} = \Delta_i'^{-*}; b \mapsto \Delta'_i(*), b$.
  Γ'_{i+1} and Δ'_{i+1} are compatible, their union is transitive and they coincide on the intersection of their domains.
  $(\Gamma'_{i+1} \cup \Delta'_{i+1})(*) = (\Gamma'_i \cup \Delta'_i)(*), b$ where b is fresh therefore $b \notin S$ and by induction hypothesis $S \cap (\Gamma'_{i+1} \cup \Delta'_{i+1})(*) = \emptyset$.
- if $\mu_i = a\langle v \rangle(b)$ the case is similar to the previous one, swapping Γ and Δ.

This ensures us that none of the μ_i are inputs or outputs on higher-order names in S. Moreover if $a \in \mathtt{dom}(\Delta)$ and $\Delta(a) \cap S \neq \emptyset$ then for all i, $a \notin (\Gamma_i \cup \Delta_i)(*)$ otherwise we would have $\emptyset \neq \Delta(a) \cap S = \Delta_i(a) \cap S \subseteq (\Gamma_i \cup \Delta_i)(*) \cap S = \emptyset$.
Therefore the trace $\mu_1 \dots \mu_n$ is also a trace of $\mathtt{Pr}^{\Delta}_{S}(P)$ under Γ. ◀

C Additional Material for Section 5

C.1 About Example 29

The process discussed in Example 29 can be seen as the π-calculus counterpart of the following ML program, which might be more readable for a reader familiar with functional languages:

```
let h = ref 0 in
let b () = h := !h +1 in
begin
   a b;
   let x = !h in
   c x;
   let y = !h in
   if x=y then loop() else ()
end
```

Re-entrant call

The following example first shows an equality similar to that of Example 29; and then develops the example so to show how re-entrant calls may sometimes break "expected" visibility boundaries.

We define two processes P_1 and P_2, which access a reference h (implemented using a first-order name), and can call a server b. When calling b, the processes first pass the name `incr` of a server that increments the value stored in h. Once the call on b is answered the processes read the value in h and make another call to b passing the identity function. After that, the processes read again the value of h. Finally, P_1 outputs the first value read in h while P_2 outputs the second one.

$$\begin{aligned} P_i \stackrel{\text{def}}{=} \overline{b}(\texttt{incr}, q_1).q_1.\, h(x_1).\, (\overline{h}\langle x_1\rangle \\ \mid\ \overline{b}(\texttt{id}, q_2).\\ q_2.\, h(x_2).\, (\overline{h}\langle x_2\rangle \\ \mid\ \overline{p}\langle x_i\rangle)) \end{aligned}$$

We first compare the processes

$$\boldsymbol{\nu} h(\overline{h}\langle 0\rangle \mid\ a(b,p).\, P_1 \mid\ Q_{\texttt{incr}} \mid\ Q_{\texttt{id}}) \quad \text{and} \quad \boldsymbol{\nu} h(\overline{h}\langle 0\rangle \mid\ a(b,p).\, P_2 \mid\ Q_{\texttt{incr}} \mid\ Q_{\texttt{id}}),$$

where $Q_{\texttt{incr}}$ (resp. $Q_{\texttt{id}}$) is the encoding of function `incr` (resp. `id`).

Here, the only way to change the value stored in h is by calling `incr`, which is not visible by the observer after the input on q_1. These two processes are bisimilar under any visibility.

Then we compare the processes

$$\begin{aligned} Q_1 &\stackrel{\text{def}}{=} \boldsymbol{\nu} h(\overline{h}\langle 0\rangle \mid\ !a(b,p).\, P_1 \mid\ Q_{\texttt{incr}} \mid\ Q_{\texttt{id}})) \\ \text{and} \quad Q_2 &\stackrel{\text{def}}{=} \boldsymbol{\nu} h(\overline{h}\langle 0\rangle \mid\ !a(b,p).\, P_2 \mid\ Q_{\texttt{incr}} \mid\ Q_{\texttt{id}}). \end{aligned}$$

We are be able to distinguish them using a re-entrant call :

$$\begin{aligned} R \stackrel{\text{def}}{=} \overline{a}(b,p).\, (b(\texttt{incr}, q).\, \overline{a}(d,r).\, (!d(-,s).\, \overline{\texttt{incr}}(t).\, t.\, \overline{s} \\ \mid\ r(n).\, \texttt{if}\ n = 1\ \texttt{then}\ \overline{q}\ \texttt{else}\ \omega) \\ \mid\ p(m)) \end{aligned}$$

The idea is to call a a second time when `incr` is still in the observer's visibility; this means that the new server passed to a has access to `incr` during both calls. With the observer R, only Q_2 will be able to send a success signal. Therefore, for any visibility function Γ such that $a \in \Gamma(*)$, Q_1 and Q_2 are not bisimilar under visibility Γ.

The processes Q_1 and Q_2 can be seen as the π-calculus counterparts of the following ML program. (This example, with re-entrant calls, has been inspired by similar examples in game semantics of functional languages):

```
let h = ref 0 in
let incr () = h := !h +1 in
let id x = x in
let a b =
   b incr;
   let x1 = !h in
   b id;
   let x2 = !h in
   xi
```

These programs can be distinguished by the following ML program that corresponds to the process R :

```
let b f =
   let d _ = f () in
   let n = a d in
   if n = 1 then () else ω
in
a b
```

The idea is the same as before: `d` calls `f` which will be `incr` in the first call to `b`; therefore at each call to `d` the reference is incremented. As a consequence, first program returns 1 while the second one returns 2.

C.2 Properties of traces: views and well-bracketing

Along the lines of the material presented in Section 4.2, we can reason about the traces produced by typable processes, based on the notion of justification. For this, we adapt Definition 22 to define views for traces. The notion of *respecting views* (Definition 23) is also adapted straightforwardly, which allows us to obtain the analogue of Lemma 24.

▶ **Lemma 38.** *Consider $\Delta, \Gamma, \sigma, \kappa, P$ such that $\langle \Delta \mid \sigma \rangle \vdash P$ and $[\Gamma \mid \kappa]$ is compatible with $\langle \Delta \mid \sigma \rangle$. Let $A = (\Gamma \cup \Delta)(*)$, then every trace of P under $[\Gamma \mid \kappa]$ respects views with starting visibility A.*

As highlighted in [3], continuation names allow us to formulate a property of well-bracketing for traces, enforced by our type system. In contrast with [3], however, we we do not impose input receptiveness on continuation names.

▶ **Definition 39** (Well-bracketing)**.** An action of the form $a\langle v \rangle(b, p)$ or $\overline{a}\langle v \rangle(b, p)$ is called a *question*. An action of the form $p\langle v \rangle(b)$ or $\overline{p}\langle v \rangle(b)$ is called an *answer*.

A trace $\mu_1 \dots \mu_n$ is *well-bracketed* if for all $i < j$, if μ_i is a question and μ_j is an answer with $\mu_i \not\frown \mu_k$ and $\mu_k \not\frown \mu_j$ for all $i < k < j$, then $\mu_i \curvearrowright \mu_j$.

A stack σ is *clean* if each name has at most one occurrence, with any tag, in σ. A trace $\mu_0 \dots \mu_n$ is a *typed trace emanating from* P_0 if there are $\Delta_1, \sigma_1, \dots, \Delta_n, \sigma_n, P_1, \dots, P_n$ such that for all $0 \leq j < n$ we are in one of the cases given by Theorem 27, which gives $\langle \Delta_{j+1} \mid \sigma_{j+1} \rangle$ when $P_j \xrightarrow{\mu_{j+1}} P_{j+1}$.

▶ **Lemma 40.** *Suppose $\langle \Delta_0 \mid \sigma_0 \rangle \vdash P_0$, where σ_0 is clean. All typed traces emanating from P_0 are well-bracketed.*

Expressive Equivalence Between Decidable Freeze and Metric Timed Temporal Logics.

Hsi-Ming Ho
University of Sussex, UK

Shankara Narayanan Krishna
Indian Institute of Technology Bombay, Mumbai, India

Khushraj Madnani
Max Planck Institute for Software Systems (MPI-SWS), Kaiserslautern, Germany
Indian Institute of Technology Guwahati, India

Rupak Majumdar
Max Planck Institute for Software Systems (MPI-SWS), Kaiserslautern, Germany

Paritosh Pandya
Indian Institute of Technology Bombay, Mumbai, India

Abstract

We demonstrate a surprising and first-of-its-kind expressive equivalence between decidable metric and freeze logics over timed words in pointwise semantics. Our main result states that Metric Interval Temporal Logic with future, past and Pnueli modalities, MITPPL, and full unilateral timed propositional temporal logic with both future and past temporal modalities, UPTL, have identical expressiveness. One of the highlights of this paper, which allows for this equivalence, is to prove that UPTL formulas admit monadic decomposition. Our result also implies that several decidable logics for real-time specifications, such as one-variable UPTL, unilateral MITPPL, and Q2MLO, are all expressively equivalent, and the reductions between them are effective. Hence, our result unifies the fragmented expressiveness boundary of timed temporal logics. As corollaries, we resolve the open question of the decidability for full UPTL, and the variable or clock hierarchy problem for the future fragment of UPTL.

2012 ACM Subject Classification Theory of computation → Logic

Keywords and phrases Timed Propositional Temporal Logic, Expressiveness, Metric Interval Temporal Logic, Satisfiability, Decidability

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2025.24

Funding This research was funded in part by the Deutsche Forschungsgemeinschaft (DFG) project 389792660 TRR 248-CPEC.

1 Introduction

Timed logics provide means to declaratively specify properties of real-time systems. They extend discrete time temporal logics such as linear temporal logic (LTL) by associating real-time intervals with temporal modalities, or with freeze quantifiers. An important research goal in the algorithmic verification of real-time systems is to balance the desire for more expressive logics with their decidability and tractability. Timed logics have two main semantics : *pointwise* (or event-based), where it is represented as a sequence of (events, timestamp) pairs called a timed word, and *continuous*, where it is represented as a signal (as a function from non-negative real numbers to a set of events).

There has been a long quest for highly expressive, yet decidable real-time temporal logics [1, 32, 9, 11, 26, 21, 24, 15, 8, 22, 23]. It is well known that timed propositional temporal logic TPTL, which extends LTL with "freeze" quantifiers to record the time elapsed [3], and

© Hsi-Ming Ho, Shankara Narayanan Krishna, Khushraj Madnani, Rupak Majumdar, and Paritosh Pandya;
licensed under Creative Commons License CC-BY 4.0

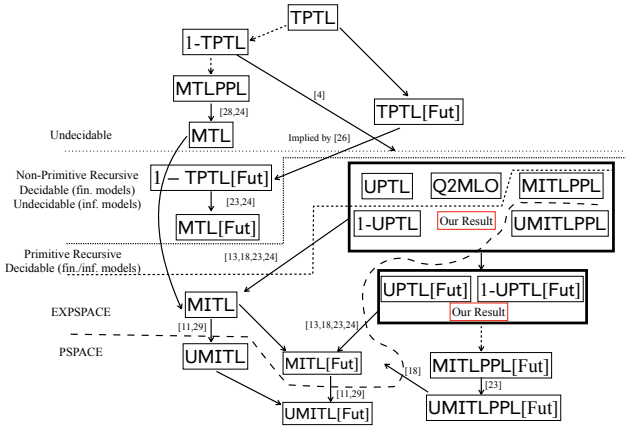


Figure 1 The expressiveness comparison (over timed words) of various timed logics studied in the literature. An arrow from L_1 to L_2 means that L_1 is more expressive than L_2. A dashed arrow between L_1 and L_2 means that strict containment wrt. expressiveness is still open. A solid arrow implies strict containment. Logics within the same box are equi-expressive. Our main contribution is to establish the equivalences shown in this figure.

metric temporal logic MTL, which extends LTL with temporal operators indexed by real-time intervals [20], are expressive but undecidable [4, 28] in both pointwise and continuous semantics. Moreover, TPTL (respectively, TPTL[Fut] the future only fragment of TPTL) is strictly more expressive than MTL (and MTL[Fut], respectively) in both pointwise and continuous semantics [5, 29]. This has led to a search for logics that restrict the expressiveness of TPTL and MTL in various ways to recover the decidability.

The continuous semantics naturally extends the first order theory of untimed words, and techniques from the untimed setting can be generalized, albeit with non-trivial extensions, such as adapting the separation lemma for expressive completeness [19]. In the pointwise semantics, these expressive completeness (and other equivalences) break down, leading to fragmented sublogics with varying expressiveness. This makes adapting traditional techniques difficult, and expressive equivalences in pointwise semantics are rare. Even though the continuous semantics is theoretically more well-behaved [12], the pointwise semantics is more amenable to practical implementations for verification applications [7, 10]. Currently, the expressiveness landscape for decidable real-time temporal logics – especially under the pointwise semantics – is quite fragmented: there are a number of different proposals that maintain decidability, but many questions remain unanswered about their relative expressive power. This paper attempts to fill in this gap.

Figure 1 shows a reasonably complete view of the current decidability/expressiveness landscape in the pointwise semantics[1]. The arrows $\mathsf{L} \to \mathsf{L}'$ in the figure indicate that logic L is strictly more expressive than logic L'. At the top of the figure sits the undecidable logics TPTL and MTL. Towards the bottom sits MITL, with an EXPSPACE-complete satisfiability problem [1, 2]. More recently, a number of other decidable logics have been proposed and studied independently. These logics either extend MITL with counting operators (Pnueli modalities) MITPPL [14, 13], or restrict TPTL to *unilateral* constraints of the form $(T-x) \leq u$ or $(T-x) \geq l$ (UPTL) [24] or define syntactic fragments of a monadic first-order logic with metric and order restricting the number of variables (Q2MLO)[11, 9, 30, 32].

[1] In this paper, we implicitly assume logics have both future and past modalities. When we restrict a logic L to only future modalities, we explicitly denote it as L[Fut]. Our results span finite and infinite words.

Our Contributions.

1. **Equivalence between Metric and Freeze timed logics**. We show a surprising expressive equivalence between the metric logic MITPPL and the freeze logic UPTL. To the best of our knowledge, this is the first expressive equivalence result for a metric temporal logic and a freeze logic which works in both pointwise and continuous semantics. The key technical ingredient that enables us to prove our main result is that the formulas of UPTL (and UPTL[Fut]) admit *monadic decomposition*. That is, every UPTL (or UPTL[Fut]) formula with multiple free variables can be reduced to a Boolean combination of formulas containing only one free variable preserving equivalence. The only other such known equivalences are in continuous semantics between TPTL[$\mathbb{Q}$] and MTL[$\mathbb{Q}$] [19], and TPTL and MTL with counting [18]. Together with known equivalences from the literature [17], this implies that the logics MITPPL, UPTL, and Q2MLO are all expressively equivalent (see the box in the middle of Figure 1).
2. **Potential Applications**. The reduction of freeze logics to metric logics has practical implications for algorithmic verification. A key feature of MITL (and MITPPL) is that the satisfaction of any subformula is "local", i.e. can be defined with respect to a single point – the point of assertion. This property simplifies compositional verification for decidable metric logics and has enabled tools like MightyL [6] for MITL[Fut] and the more recent MightyPPL [27] for MITPPL. Since our reduction translates freeze logics into metric logics, it facilitates the use of these tools for verifying UPTL properties, broadening their applicability in practice.
3. **Refining the Expressiveness Boundary of Freeze Timed Logics**. We address open problems from [24] regarding the decidability of the full logic UPTL (with future and past modalities). In particular, we show that (1) the satisfiability and model-checking problems for UPTL are decidable, and (2) UPTL has the same expressive power as its one-variable fragment, indicating that increasing the number of variables in UPTL does not enhance expressiveness.

2 Preliminaries

Let $\mathbb{R}_{\geq 0}$, $\mathbb{N}$, $\mathbb{N}_{>0}$, $\mathbb{Q}$, $\mathbb{Q}_{\geq 0}$, and $\mathbb{Z}$ respectively represent the set of non-negative reals, naturals (including 0), naturals (excluding 0), rationals, non-negative rationals, and integers. Let $\langle$ denote left open "(" or left closed "[", and $\rangle$ denote right open ")" or right closed "]", and let $\mathbb{I}$ denote the set of all intervals $\langle l, u\rangle$ for $l \leq u$, $l \in \mathbb{N}$, $u \in \mathbb{N} \cup \{\infty\}$. Let true and false be denoted by $\top$ and $\bot$, respectively. Given any sequence $\eth = a_1, a_2, \ldots,$, $|\eth|$ denote length of the sequence.

Let AP be a finite set of propositional variables, and let $\Sigma_{\mathsf{AP}} = 2^{\mathsf{AP}}$ be the finite set containing all subsets of AP. An infinite timed word (or just timed words) ρ over Σ_{AP} is an infinite sequence of pairs of letters and non-decreasing time stamps of the form $\rho = (\sigma_1, \tau_1)(\sigma_2, \tau_2)\ldots$ where for all $i > 0$, $\sigma_i \in \Sigma_{\mathsf{AP}}, \tau_i \in \mathbb{R}_{\geq 0}$, and $0 \leq \tau_i \leq \tau_{i+1}$. A finite timed word is a finite sequence of pairs of letters and non-decreasing time stamps.

For $i > 0$, we define $\rho[i] = \sigma_i$ for the element of Σ_{AP} at the i^{th} position of ρ. The set of all timed words over Σ is denoted $T\Sigma^\omega$. A timed word is called *Zeno* if the sequence $(\tau_i)_{i\geq 0}$ converges, and non-Zeno otherwise. We restrict ourselves to non-Zeno infinite timed words (henceforth called timed words), which is the usual convention, as Zeno words allow infinite actions within a finite duration, which does not model a natural behaviour.

2.1 Metric Temporal Logic with Pnueli Modalities, MTLPPL

Formulas of MTLPPL are defined over a set of propositional variables AP as follows:
$\varphi := a \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi\, \mathbf{U}_I\, \varphi \mid \varphi\, \mathbf{S}_I\, \psi \mid \mathbf{Pn}_I(\varphi_1, \ldots, \varphi_k) \mid \overleftarrow{\mathbf{Pn}}_I(\varphi_1, \ldots, \varphi_k)$
where $a \in \mathsf{AP}$ and I is an interval in $\mathbb{I}$. Given a timed word $\rho = (\sigma_1, \tau_1)(\sigma_2, \tau_2)\ldots$ over Σ_{AP}, a position $i \in \mathbb{N}_{>0}$, and any $a \in \mathsf{AP}$, we define the *pointwise-semantics* of MTLPPL inductively. We define $\rho, i \models a$ iff $a \in \sigma_i$, and conjunction, disjunction, and negation in the usual way. For temporal modalities, we define:

- $\rho, i \models \varphi_1\, \mathbf{U}_I\, \varphi_2$ iff $\exists j > i$ s.t. $\tau_j - \tau_i \in I$, $\rho, j \models \varphi_2$, and $\forall i < k < j, \rho, k \models \varphi_1$;
- $\rho, i \models \varphi_1\, \mathbf{S}_I\, \varphi_2$ iff $\exists j < i$ s.t. $\tau_i - \tau_j \in I$, $\rho, j \models \varphi_2$, and $\forall j < k < i, \rho, k \models \varphi_1$;
- $\rho, j \models \mathbf{Pn}_I(\varphi_1, \ldots, \varphi_k)$ iff $\exists i_k > i_{k-1} > \ldots > i_1 > j$ s.t. $\forall 1 \le n \le k, \tau_{i_n} - \tau_j \in I$, and $\rho, i_n \models \varphi_n$;
- $\rho, j \models \overleftarrow{\mathbf{Pn}}_I(\varphi_1, \ldots, \varphi_k)$ iff $\exists i_k < i_{k-1} < \ldots < i_1 < j$ s.t. $\forall 1 \le n \le k, \tau_j - \tau_{i_n} \in I$, and $\rho, i_n \models \varphi_n$.

We define the language of φ as $[\![\varphi]\!] = \{(\rho, i) | \rho, i \models \varphi\}$. Notice that we use strict semantics for both $\mathbf{U}$ and $\mathbf{S}$, as it can easily express "next" and "previous" modalities, unlike the non-strict variants. Derived operators $\Diamond_I$ (eventually), $\Box_I$ (globally), $\overleftarrow{\Diamond}_I$ (past), $\overleftarrow{\Box}_I$ (globally in the past), O (next) and $\overleftarrow{\mathsf{O}}$ (previous) are written in terms of $\mathbf{U}$ and $\mathbf{S}$ as follows. $\Diamond_I \varphi = \top\, \mathbf{U}_I\, \varphi$, $\Box_I \varphi = \neg \Diamond_I \neg\varphi$, $\overleftarrow{\Diamond}_I \varphi = \top\, \mathbf{S}_I\, \varphi$, $\overleftarrow{\Box}_I \varphi = \neg \overleftarrow{\Diamond}_I \neg\varphi$, $\mathsf{O}\varphi = \bot\, \mathbf{U}_{[0,\infty)}\, \varphi$ and $\overleftarrow{\mathsf{O}}\, \varphi = \bot\, \mathbf{S}_{[0,\infty)}\, \varphi$. For the sake of succinctness, we omit the subscript when the intervals are $[0, \infty)$. Hence, $\varphi\, \mathbf{U}_{[0,\infty)}\, \psi$ and $\varphi\, \mathbf{S}_{[0,\infty)}\, \psi$ are written as $\varphi\, \mathbf{U}\, \psi$ and $\varphi\, \mathbf{S}\, \psi$, respectively. We define some important subclasses of MTLPPL. The subclass *Metric Interval Temporal Logic with Pnueli modalities*, written MITPPL, consists of all MTLPPL formulas where the intervals I are non-singular (i.e., of the form $\langle l, u\rangle$, where $l < u$). Metric Temporal Logic (MTL) and Metric Interval Temporal Logic (MITL) are fragments of MTLPPL and MITPPL, respectively, without Pnueli modalities. Unilateral fragments of MITL and MITPPL (denoted by UMITL and UMITPPL, respectively) are subclasses where the intervals are either of the form $\langle 0, u\rangle$ or $\langle l, \infty)$.

Example. For the UMITPPL formula $\varphi = \mathbf{Pn}_{(0,2)}(a, b, c)$, $\rho, 1 \models \varphi$ for $\rho = (\{a\}, 0)(\{a\}, .5)$ $(\emptyset, .9)$ $(\{b, c\}, 1.1)(\{a, b, c\}, 1.8)\ldots$, since $\tau_5 - \tau_1 = 1.8, \tau_4 - \tau_1 = 1.1, \tau_2 - \tau_1 = 0.5$ all lie in $(0, 2)$, and $\rho, 2 \models a, \rho, 4 \models b$ and $\rho, 5 \models c$. However, $\rho', 1 \not\models \varphi$ for $\rho' = (\{a\}, 0)(\{c\}, .1)$ $(\{b, c\}, 1.1)(\{a, b\}, 1.9)(\emptyset, 2)\ldots$.

2.2 Timed Propositional Temporal Logic, TPTL

TPTL [4] is an extension of linear temporal logic (LTL) with real-valued variables and *freeze quantifiers* which allow subformulas to freeze the timestamps of the present point to a particular variable. We define the TPTL formulas in negation normal form, i.e. where the negation operator is only applied to atomic formulas. It is well known [24] that any formula in TPTL can be converted to an equivalent formula in negation normal form (NNF), and as such, we introduce TPTL in NNF, where all negations appear only at the level of propositional variables. Formulas in TPTL (in negation normal form) over a finite set X of freeze variables and AP is defined as[2]:
$\varphi := a \mid \neg a \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid x.\varphi \mid (T - x) \sim c \mid \varphi \mathbf{U} \varphi \mid \varphi \mathbf{S} \psi \mid \Box\varphi \mid \overleftarrow{\Box}\varphi$, $a \in \mathsf{AP}$, $x \in X$, $c \in \mathbb{Z}$.

Given a timed word $\rho = (\sigma_1, \tau_1)(\sigma_2, \tau_2)\ldots$ over Σ_{AP}, a position $i \in \mathbb{N}_{>0}$, a valuation $\nu : X \to \mathbb{R}_{\ge 0}$ of the variables, and any $a \in \mathsf{AP}$, we define the *pointwise-semantics* of TPTL inductively. Propositions $a \in \mathsf{AP}$, conjunctions, disjunctions, $\mathbf{U}$, $\mathbf{S}$, $\Box$, and $\overleftarrow{\Box}$ formulas are evaluated as in LTL. In addition,

[2] In literature, the negation normal form usually admits a "Release" modality (and its past counterpart), but they can be expressed using "$\mathbf{U}$" and "$\Box$" (respectively, $\mathbf{S}$ and $\overleftarrow{\Box}$) without negations. Moreover, we use strict versions of $\mathbf{U}$ and $\mathbf{S}$ modalities.

- $\rho, i, \nu \models x.\varphi$ iff $\rho, i, \nu' \models \varphi$, where $\nu' = \nu[x \leftarrow \tau_i]$ is a valuation s.t. $\nu'(y) = \nu(y)$ for $y \neq x$, and $\nu'(x) = \tau_i$. Further, $x.\varphi$ is called a *freeze quantifier*. The timestamp τ_i of ρ is "frozen" in $x \in X$ before evaluating φ.
- $\rho, i, \nu \models (T - x) \sim c$ iff $\tau_i - \nu(x) \sim c$. Intuitively, at the ith position of ρ, the time elapsed since the last time x was frozen (i.e. $\tau_i - \nu(x)$) is compared with $c \in \mathbb{Z}$. If the last time x was frozen happens to be at position $j \leq i$, then $\nu(x) = \tau_j$ and $\tau_i - \tau_j$ is the time elapse since freezing x.

We denote by k-TPTL the class of TPTL formulas with k freeze variables; so TPTL=$\bigcup_{k\geq 0} k$-TPTL. The derived modalities next (O), eventually ($\Diamond$), sometime ago ($\overleftarrow{\Diamond}$), and previous ($\overleftarrow{\mathsf{O}}$) are defined as usual.

Open and Closed TPTL Formulas. A constraint $(T - x) \sim c$ appearing in a TPTL formula φ is called *closed* if it lies within the scope of a freeze variable $x.$, and *open* otherwise. A freeze variable x in a TPTL formula φ is called a *free variable* if any occurrence of x in φ appears in an open constraint. We write $\varphi(x_1, \ldots, x_k)$ to denote that the free variables in φ are $x_1, \ldots, x_k$. When the free variables in a formula are clear from the context, we just write φ for $\varphi(x_1, \ldots, x_k)$. A TPTL formula is *closed* iff it does not contain any free variable; an *open* formula is one which is not closed.

▶ **Example.** *The formula* $\psi = x.((a \wedge (T - z) \leq 2)\ \mathbf{U}\ [b \wedge y.(\Diamond a \wedge (T - y) < 1 \wedge (T - x) < 2)])$ *is open.* $\varphi_{cnt} = x.(\Diamond(a \wedge \Diamond(a \wedge (T - x) < 1)))$ *is closed.*

The satisfaction of a closed TPTL formula is independent of any valuation ν of its variables since each constraint $(T - x) \sim c$ lies in the scope of a freeze quantifier $x.$, the value of x only depends on the time elapsed since it was frozen. Hence, we define the language of a TPTL formula φ only if it is closed. We define the language of a closed formula φ as $[\![\varphi]\!] = \{(\rho, i) | \rho, i \models \varphi\}$. The formula φ_{cnt} above encodes the property that there are at least two a in the next time interval of unit length; it is known [11] that φ_{cnt} does not have an equivalent MTL formula. Then $(\rho, 1) \in [\![\varphi_{cnt}]\!]$ where $\rho = (\{a\}, 0.2)(\{a, b\}, 0.45)(\emptyset, .75)(\{b\}, .8)(\{c\}, .95) \ldots$.

Polarity of a TPTL formula. Let $(T - x) \sim c$ be an open constraint in formula φ. The *polarity* of $(T - x) \sim c$ denoted Pol$((T - x) \sim c)$ is defined to be the singleton set $\{\leq\}$ if $\sim \in \{<, \leq\}$, and $\{\geq\}$ if $\sim \in \{>, \geq\}$. The polarity of a TPTL formula φ denoted Pol(φ) is defined when all open constraints in φ have the *same* polarity. Pol(φ) is the polarity of the open constraints in φ. We define the polarity of a closed TPTL formula as the set $\{\leq, \geq\}$ since it has no open constraints.

Unilateral Timed Propositional Temporal Logic, UPTL. A TPTL formula φ is UPTL iff Pol(ψ) is defined for all subformulas ψ of φ. Let k-UPTL denote UPTL formulas using k freeze variables, and UPTL=$\bigcup_{k\geq 0} k$-UPTL.

▶ **Example.** *Consider* $\varphi = x.[\Diamond(a \wedge (T - x) > 2 \wedge (T - y) > 3)]$. Pol$((T - y) > 3) = \{\geq\} =$ Pol(φ). *Likewise,* Pol$(\kappa) = \{\leq\}$ *for* $\kappa = x.(\Diamond(a \wedge (T - x) > 2 \wedge (T - y) < 3))$. *However,* Pol$(\psi)$ *is undefined for the subformula* $\psi = \Diamond(a \wedge (T - x) > 2 \wedge (T - y) < 3)$ *of* κ*, since* Pol$((T - x) > 2) = \{\geq\}$ *while* Pol$((T - y) < 3) = \{\leq\}$.

Future Fragments of Logics. For any logic L defined so far, L[Fut] denotes its future fragment, one which does not involve any past (or past Pnueli) modalities. For instance, UPTL[Fut] is the fragment of UPTL with only future modalities [24].

▶ Remark 2.1. Notice that MTLPPL (and UMITPPL) properties can be translated to 1-TPTL (1-UPTL) preserving the languages by applying the following reduction inductively. $\mathbf{Pn}_I(a_1,\ldots,a_k) \equiv x.(\Diamond(a_1 \wedge (T-x)\in I \wedge \Diamond(a_2 \wedge (T-x)\in I \wedge \Diamond(\ldots\Diamond(a_k \wedge (T-x)\in \overleftarrow{I})\ldots)$. Moreover, $a_1\ \mathbf{U}_I\ a_2 \equiv x.(a_1\ \mathbf{U}\ (a_2 \wedge (T-x)\in I))$. Similar reductions apply for $\mathbf{S}$ and $\overleftarrow{\mathbf{Pn}}$.

2.3 First Order Logic of Metric and Order, FOMLO

Formulas of FOMLO are defined over a signature containing *monadic* predicates $a(x)$ where $a \in \mathsf{AP}$, *order* predicates $x \sim y$ for $\sim \in \{<,>,=,\leq,\geq\}$ and a set of *metric* predicates $\Delta = \{(x \ominus y) \sim c \mid c \in \mathbb{Z}, \sim \in \{\leq,\geq,<,>,=\}\}$, where x, y are first-order variables. Syntax of FOMLO formula $\varphi(x_1,\ldots,x_k)$ is defined as:
$\varphi(x_1,\ldots,x_k) :: a(x_i) \mid x_i \sim x_j \mid (x_i \ominus x_j)\sim c \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \neg\varphi \mid \mathcal{Q}x.\varphi(x,x_1,\ldots,x_k)$,
where $\mathcal{Q} \in \{\exists,\forall\}$ and is interpreted over timed words ρ along with valuation $\nu : \{x_1,\ldots,x_k\} \to \mathbb{N}_{>0}$. $\rho,\nu \models (x \ominus y)\sim c$ for $\nu(x) = j, \nu(y) = i$ iff $\tau_j - \tau_i \sim c$, where τ_j and τ_i are timestamps of points i, j in ρ, respectively. Hence, the variables range over positions of timed words. The semantics of the remaining operators are identical to that of first-order logic with $<$ predicate (FO[$<$]) over words(see [31][11] for more details). For a FOMLO formula $\varphi(x)$ with one free variable, the language $[\![\varphi(x)]\!]$ is defined as $\{(\rho,i) \mid \rho,\nu \models \varphi(x)$ with $\nu(x) = i\}$.

Q2MLO [11] is a syntactic fragment of FOMLO where, in the grammar of FOMLO, metric predicates $(x \ominus y) \sim c$ are replaced by the timed monadic formula $\psi(x)$ whose syntax is $\psi(x) := \exists y.[(y \ominus x < u) \wedge (y \ominus x > l) \wedge \varphi(x,y)]$ where $l, u \in \mathbb{Z}$, $l \neq u$, and $\varphi(x,y)$ is a well-formed Q2MLO formula. Q2MLO can be seen as an extension of FO[$<$] with timed monadic formulas.

Example. The formula $\exists y.\exists z.[x>y>z \wedge a(y) \wedge b(z) \wedge (z \ominus x) > 15 \wedge (z \ominus y) < 30]$
is in FOMLO but not in Q2MLO, as there is a subformula containing time constraints involving two free variables.

2.4 Expressive Equivalence

Let ϕ and ψ be in TPTL. Then ϕ is said to be equivalent to ψ iff for any timed word ρ, any point i of ρ, and any valuation ν over free variables of ϕ and ψ, $\rho,i,\nu \models \phi$ iff $\rho,i,\nu \models \psi$. The equivalence between two Q2MLO formulas is defined in the classical sense. We say that two formulas φ and ψ are *language equivalent* iff $[\![\varphi]\!] = [\![\psi]\!]$. Logic L_1 is at least as expressive as L_2 (denoted by $\mathsf{L}_2 \preceq \mathsf{L}_1$) iff for any formula φ in L_1, there exists a language equivalent ψ in L_2. L_1 and L_2 are said to be equally expressive (denoted by $\mathsf{L}_1 \equiv \mathsf{L}_2$) iff $\mathsf{L}_1 \preceq \mathsf{L}_2$ and $\mathsf{L}_2 \preceq \mathsf{L}_1$. Finally, L_1 is said to be strictly more expressive than L_2 (denoted by $\mathsf{L}_2 \prec \mathsf{L}_1$) iff $\mathsf{L}_2 \preceq \mathsf{L}_1$, and L_2 and L_1 are not equally expressive.
Extension to rational constants. For any logic L, let L[$\mathbb{Q}$] denote the extension of L where the constants used in the timing constraints are from the set $\mathbb{Q}_{\geq 0}$. For example, $a\ \mathbf{U}_{[0,\frac{1}{2}]}\ b$ is not in MITL but is in MITL[$\mathbb{Q}$].

3 Main Results

In this section, we discuss our main result (Theorem 3.1) connecting UPTL and MITPPL. A "guided tour" to navigate supporting results for each theorem is given in Figure 2.

▶ **Theorem 3.1.**
(1) UPTL ≡ 1-UPTL ≡ MITPPL ≡ UMITPPL ≡ Q2MLO.
(2) MITL[$\mathbb{Q}$]≡UPTL[$\mathbb{Q}$]≡1-UPTL[$\mathbb{Q}$]≡MITPPL[$\mathbb{Q}$]≡UMITPPL[$\mathbb{Q}$]≡Q2MLO[$\mathbb{Q}$].

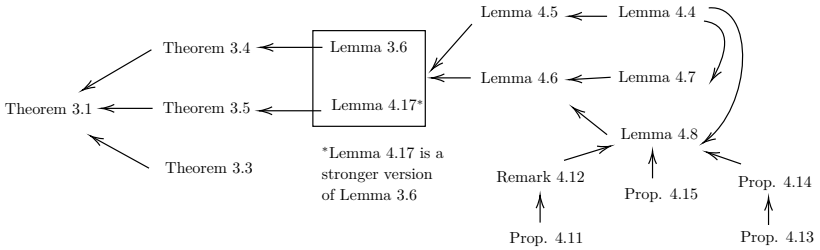


Figure 2 Dependencies between theorems and lemmas. $A \leftarrow B$ means proof of A uses B.

▶ Remark 3.2. Since MITPPL has no freeze variables, we compare it with closed UPTL formulas, similar to how Kamp's theorem compares LTL with FO[<] having one free variable. Our equivalence between 1-UPTL and UPTL also holds for open formulas. A reduction from open 1-UPTL formulas to an extension of Q2MLO with two free variables also holds, but it would lengthen the proof unnecessarily.

3.1 Overview of the proof of Theorem 3.1

Theorem 3.1 is a direct consequence of Theorem 3.3, Theorem 3.4 and Theorem 3.5 below. Theorem 3.3 below is a result of [16, 17].

▶ **Theorem 3.3** ([16, 17])**.**
(1) UMITPPL $\equiv$ MITPPL $\equiv$ Q2MLO.
(2) MITL[$\mathbb{Q}$] $\equiv$ UMITPPL[$\mathbb{Q}$] $\equiv$ MITPPL[$\mathbb{Q}$] $\equiv$ Q2MLO[$\mathbb{Q}$].
Thanks to the above theorem, we just need to show that UPTL $\equiv$ 1-UPTL $\equiv$ Q2MLO, and UPTL[$\mathbb{Q}$] $\equiv$ 1-UPTL[$\mathbb{Q}$] $\equiv$ Q2MLO[$\mathbb{Q}$]. The former is implied by two of our main results, namely Theorems 3.4 and 3.5. The latter is implied by the fact that the proofs of Theorems 3.4 and 3.5 do not assume the constants to be integers.

▶ **Theorem 3.4.** UPTL $\equiv$ 1-UPTL.

▶ **Theorem 3.5.** 1-UPTL $\equiv$ Q2MLO.

Theorems 3.4 and 3.5 rely on the following main technical lemma, Lemma 3.6.

▶ **Lemma 3.6** (Monadic Decomposition of UPTL)**.** *There is an effective translation from any* $\varphi(x_1, \ldots, x_k) \in$ UPTL *to an equivalent* UPTL *formula* ψ*, where* ψ *is a Boolean combination of* UPTL *formulas with at most one free variable.*

We first discuss the proof of Theorem 3.4 assuming Lemma 3.6. The proof of Lemma 3.6 and Theorem 3.5 are deferred to sections 4, 5, respectively.

3.2 Proof of Theorem 3.4

1-UPTL $\preceq$ UPTL follows trivially from the fact that the former is a syntactic fragment of the latter. Hence, it suffices to show that UPTL $\preceq$ 1-UPTL. Let φ be any closed UPTL formula using variables $x_1, \ldots, x_k$. We construct a closed 1-UPTL formula ψ which uses only a single clock x using the following steps.

1. **Start with the innermost freeze quantifier over a variable,** x_i. Consider a subformula of the form $x_i.\varphi'(x_1, \ldots, x_k)$, such that φ' does not contain any subformula of the form $x_j.\varphi''$, where $x_i, x_j \in \{x_1, \ldots, x_k\}$.

2. **Monadically decompose the formula** φ'. By Lemma 3.6, we can construct an equivalent formula ψ' which is a Boolean combination of formulas of the form $\psi_j^m(x_j)$.
3. **Distribute the freeze quantifier** x_i. Freeze quantifiers distribute over $\wedge$ and $\vee$. Hence, $x_i.\varphi'(x_1,\ldots,x_k)$ can be equivalently written as the Boolean combination of formulas of the form $x_i.\psi_j^m(x_j)$. Moreover, $x_i.\psi_j^m(x_j)$ is equivalent to (i) $\psi_j^m(x_j)$ when $i{\neq}j$, and (ii) to $x.\psi_j^m(x)$ when $i=j$. Hence, we either remove the freeze quantifier $(x_i.)$ in case $i{\neq}j$, or rename the variable x_i to x. This step strictly reduces the number of freeze quantifiers over some variable $x_i \in \{x_1,\ldots,x_k\}$.
4. Repeat steps 1, 2, and 3 until we remove all the freeze quantifiers over variables $\{x_1,\ldots,x_k\}$. Hence, after performing the above steps at most n times, where n is the number of freeze quantifiers in φ, we get an equivalent formula ψ containing only one variable x. ◀

4 Proof of Lemma 3.6: Monadic Decomposition of UPTL

The main ingredient to achieve monadic decomposition is to come up with a novel syntactic subclass of UPTL called *Conjunctive Timed Clauses* (CoT, defined later) which has the following properties:

P1 CoT formulas are "easily" monadically decomposable;

P2 Any $\varphi \in$ UPTL can be expressed as a Boolean combination of CoT formulas.

Notice that Lemma 3.6 follows trivially if we can prove **P1** and **P2**.

▶ **Definition 4.1.** *(Conjunctive Timed Clauses (CoT)) A* UPTL *formula is a* conjunctive timed clause *(CoT) if it is generated by the following grammar:*

$$\psi ::= \phi \mid (T-x) \sim c \mid \psi \wedge \psi \mid \phi \,\mathbf{U}\, \psi \mid \phi \,\mathbf{S}\, \psi$$

where ϕ is any closed UPTL *formula, $x \in X$ is a freeze variable, and $c \in \mathbb{Z}$.*

Note that all open time constraints in CoT formulas appear on the right side of **U** and **S** modalities and/or conjunctions. Intuitively, we make sure that all the open time constraints are existentially quantified, and do not appear within the scope of a disjunction. Further, $\Diamond(\psi)$, $\overleftarrow{\Diamond}(\psi)$, $\mathsf{O}(\psi)$, and $\overleftarrow{\mathsf{O}}\,(\psi)$ are valid production rules as they are restricted forms of **U** and **S**. Moreover, $\Box, \vee$ formulas appears in a CoT only in ϕ, and hence all instances of $\Box, \vee$ in a CoT are within a closed subformula.

Example. $x.\Box(\Diamond((T-x)<2))$ is a well-formed CoT formula but $\Box(\Diamond((T-x)<2))$ is not. $\phi = \mathsf{O}((T-x)<1 \wedge (T-y)<3) \wedge [\{\Box(z.\Diamond(b \vee (T-z)<3))\}\ \mathbf{U}\ \{(T-x)<1\}]$ is in CoT. However, $\psi = (b \wedge (T-y)<1)\ \mathbf{U}\ [x.(\Diamond(a \wedge (T-x)<2))]$ is not in CoT; even though $x.(\Diamond(a \wedge (T-x)<2))$ is in CoT, $(b \wedge (T-y)<1)$ is not closed.

Untiming Property. To show that CoT satisfies properties **P1** and **P2**, we rely on a semantic property of CoT: they can be "untimed". Intuitively, formulas in CoT of polarity $\{\leq\}$ (resp. $\{\geq\}$) enjoy a certain form of "downward closure" (resp. "upward closure") w.r.t. their satisfaction by removing the open constraints appearing in them. This is described by the *Untiming Lemma* (Lemma 4.4), the name suggestive of the fact that they can be untimed by removing the open constraints under certain conditions. Given a TPTL formula ψ, let Untime(ψ) denote the set of all formulas obtained by substituting $\top$ for some/all/none of the open time constraints occurring in ψ.

Example. For $\psi{=}(T-x){<}u_1 \wedge \Diamond((T-y){<}u_2) \wedge x.\Diamond((T-x){<}u_3)$, Untime($\psi$)$=\{\psi,\ \Diamond((T-y){<}u_2) \wedge x.\Diamond((T-x){<}u_3),\ (T-x){<}u_1 \wedge x.\Diamond((T-x){<}u_3),\ x.\Diamond((T-x){<}u_3)\}$.

▶ **Definition 4.2** (The Untiming Property). *Let* $\bowtie \in \{\leq, \geq\}$. *A* TPTL *formula* $\psi(x_1, \ldots, x_k)$ *satisfies the* $\bowtie$ untiming property *iff for any* $\psi' \in \mathsf{Untime}(\psi)$, *a timed word* ρ, *a position* $j \in \mathbb{N}_{>0}$, *and a valuation* $\nu : \{x_1, \ldots, x_k\} \mapsto \mathbb{R}_{\geq 0}$, *we have* $\rho, j, \nu \models \psi$ *implies for all* $i \in \mathbb{N}_{>0}$ *with* $i \bowtie j$ *we have* $\rho, i, \nu \models \psi'$ *iff* $\rho, i, \nu \models \psi$.

▶ Remark 4.3. A formula of the form $\mathsf{CoT} \vee \mathsf{CoT}$ may not satisfy the untiming property. Consider $\psi_{ex} = a \vee (b \wedge ((T - x) \leq 2))$, $\mathsf{Pol}(\psi_{ex}) = \{\leq\}$, and $a \vee b$ is in $\mathsf{Untime}(\psi_{ex})$. Let ρ be any timed word, and j be a point in ρ where a holds. Then, $\rho, j, \nu \models \psi_{ex}$ for any clock valuation ν. However, for any point $i < j$ and any valuation ν over x, $\rho, i, \nu \models a \vee b$ iff $\rho, i, \nu \models \psi_{ex}$ need not be true; for instance, consider $b \wedge \neg a$ is true at i, $\tau_i = 50$ and $\nu(x) = 1$.

▶ **Lemma 4.4** (The Untiming Lemma). *Let* $\bowtie \in \{\leq, \geq\}$. *Any* CoT *formula* ψ *of polarity* $\bowtie$ *satisfies the* $\bowtie$ *untiming property.*

Proof. Let ψ be a formula in CoT. We prove for the case $\leq \in \mathsf{Pol}(\psi)$; a symmetric argument follows when $\geq \in \mathsf{Pol}(\psi)$. Fix a timed word $\rho = (a_1, \tau_1)(a_2, \tau_2)\ldots$, a point j of ρ and a valuation ν over the variables in ψ. Let $\psi' \in \mathsf{Untime}(\psi)$. Assuming that (A) $\rho, j, \nu \models \psi$, we now show that (B) for any point $i \leq j$, $\rho, i, \nu \models \psi$ iff $\rho, i, \nu \models \psi'$. We apply induction on the structure of the formula ψ. From now, we assume that (A) holds. Notice that (B-1) implies (B-2) holds trivially. This is because all the negations in the formula ψ apply only to propositional variables. Hence, all the timing constraints appear positively within ψ. Thus, replacing some/all timing constraints by true should make satisfaction easier. Hence, it suffices to show that for any point $i \leq j$, (B-2) implies (B-1).

Base Case. Let ψ be either a closed formula or an atomic clock constraint of the form $(T - x) \sim u$ where $\sim \in \{<, \leq\}$. If ψ is a closed formula then $\mathsf{Untime}(\psi) = \{\psi\}$. Hence, the lemma holds trivially. If $\psi = (T - x) \sim u$, then $\mathsf{Untime}(\psi) = \{\psi, \top\}$. For any $\psi' \in \mathsf{Untime}(\psi)$, (B-1) implies (B-2) is a tautology as discussed above. To show that for all $i \leq j$ (B-2) implies (B-1), we need to show that $\rho, i, \nu \models \top$ implies $\rho, i, \nu \models (T - x) \sim u$ for all $i \leq j$. Note that for any $i \leq j$, $\tau_i \leq \tau_j$. As (A) holds, we know $\tau_j - \nu(x) \sim u$; hence for any $i \leq j$, $\tau_i - \nu(x) \leq \tau_j - \nu(x) \sim u$, giving $\rho, i, \nu \models (T - x) \sim u$.

Induction Hypothesis. Assume the lemma holds for all strict subformulas of ψ.

Induction. We now show that given (A) and induction hypothesis for any $i \leq j$ (B-2) implies (B-1) given the induction hypothesis and (A).

There are three possible cases (1) $\psi = \psi_1 \wedge \psi_2$, (2) $\psi = \psi_1 \,\mathbf{U}\, \psi_2$, and (3) $\psi = \psi_1 \,\mathbf{S}\, \psi_2$. If ψ is an open formula, then it cannot be of the form $\psi_1 \vee \psi_2$ or $\Box(\psi_1)$ by definition of CoT.

We only discuss case (2), with case (3) being symmetric. Details of all cases are in Appendix A.

Case (2). $\psi = \psi_1 \,\mathbf{U}\, \psi_2$ (*Argument Sketch*). As ψ is in CoT, ψ_1 is a closed formula. Hence, $\mathsf{Untime}(\psi_1) = \{\psi_1\}$, and ψ' is of the form $\psi_1 \,\mathbf{U}\, \psi_2'$ where $\psi_2' \in \mathsf{Untime}(\psi_2)$. For any $\psi_2' \in \mathsf{Untime}(\psi_2)$, we now show that for any $i \leq j$, (B-2) $\rho, i, \nu \models \psi_1 \,\mathbf{U}\, \psi_2'$ implies (B-1)$\rho, i, \nu \models \psi_1 \,\mathbf{U}\, \psi_2$.

Intuitively, (A) implies that there is a point $j_{min} > j$ such that $\rho, j_{min}, \nu \models \psi_2$, and all the points between j and j_{min} satisfy ψ_1. Notice, by induction hypothesis, $\rho, j_{min}, \nu \models \psi_2'$. Moreover,(B-2) implies that there exists a point $i_m > i$ such that $\rho, i_m, \nu \models \psi_2'$ and all the points between i and i_m satisfy ψ_1. Either $i_m < j_{min}$, in which case, by induction hypothesis $\rho, i_m, \nu \models \psi_2'$ iff $\rho, i_m, \nu \models \psi_2$, and hence (B-1) holds. Or, $i_m > j_{min} > i$ such that $\rho, j_{min}, \nu \models \psi_2'$ (recall by induction hypothesis), and all the points between i and i_m (thus between i and j_{min}) satisfy ψ_1. Hence, $\rho, i, \nu \models \psi_1 \,\mathbf{U}\, \psi_2$ which is equivalent to (B-1). ◀

The following sections 4.1 and 4.2 show that the subclass CoT of UPTL satisfies the properties **P1** and **P2**, respectively.

4.1 Monadic Decomposition of CoT formulas

In this section, we show that CoT admits monadic decomposition, i.e. satisfies property **P1**. Let ψ be any TPTL formula. Let $c_j = (T - x_{i_j}) \sim u_{i_j}$, $1 \leq j \leq k$ be the open constraints appearing in ψ. Let $\psi{\downarrow_{c_j}}$ be the formula obtained from $\psi(x_{i_1}, \ldots, x_{i_k})$ by substituting "true" for all open constraints except c_j with $\top$. Hence, $\psi{\downarrow_{c_j}}$ for any $1 \leq j \leq k$ can have at most one open constraint, and thus one free variable x_{i_j}.

▶ **Lemma 4.5.** *Any formula ψ in* CoT *is equivalent to $\psi{\downarrow_{c_1}}(x_{i_1}) \wedge \ldots \wedge \psi{\downarrow_{c_k}}(x_{i_k})$.*

Proof. Let $\psi(x_1, \ldots, x_k)$ be a UPTL formula in CoT. We prove for $\leq \in \mathsf{Pol}(\varphi)$, and the argument is symmetric for $\geq \in \mathsf{Pol}(\varphi)$. The proof follows by structural induction on ψ.

Base Case. If ψ is a closed formula or a conjunction of atomic time constraints, then $\psi{\downarrow_{c_1}}(x_{i_1}) \wedge \ldots \wedge \psi{\downarrow_{c_k}}(x_{i_k})$ is the same as ψ. Hence, the lemma trivially holds.

Induction Hypothesis. Let the lemma hold for all the strict subformulas of ψ.

Induction. There are three possible cases.

- $\psi = \alpha \wedge \beta$. Notice that $\alpha{\downarrow_{c_j}} \wedge \beta{\downarrow_{c_j}} = \psi{\downarrow_{c_j}}(x_{i_j})$ for all $1 \leq j \leq k$. By induction hypothesis, $\alpha \equiv \alpha{\downarrow_{c_1}} \wedge \alpha{\downarrow_{c_2}} \wedge \ldots \wedge \alpha{\downarrow_{c_k}}$, and $\beta \equiv \beta{\downarrow_{c_1}} \wedge \beta{\downarrow_{c_2}} \wedge \ldots \wedge \beta{\downarrow_{c_k}}$. Hence, $\psi = \psi{\downarrow_{c_1}}(x_{i_1}) \wedge \ldots \wedge \psi{\downarrow_{c_k}}(x_{i_k})$.
- $\psi = \Box(\alpha)$ or $\psi = \overleftarrow{\Box}(\alpha)$. Then ψ is closed as it is in CoT, and the lemma trivially holds (implied by the base case).
- $\psi = \alpha \,\mathbf{U}\, \beta$. A symmetric argument holds for $\alpha \,\mathbf{S}\, \beta$. For an overall idea, assume that ψ has only two open constraints c_1, c_2 (this argument easily generalizes for more open constraints, see Appendix B). By induction hypothesis, $\psi \equiv \psi'$, where $\psi' = \alpha \,\mathbf{U}\, (\beta{\downarrow_{c_1}} \wedge \beta{\downarrow_{c_2}})$. As α is a closed formula, for any $j \in \{1, 2\}$, $\psi{\downarrow_{c_j}} = \alpha \,\mathbf{U}\, \beta{\downarrow_{c_j}}$. We need to show that $\psi' \leftrightarrow \psi{\downarrow_{c_1}} \wedge \psi{\downarrow_{c_2}}$ is a tautology. $\psi' \rightarrow \psi{\downarrow_{c_1}} \wedge \psi{\downarrow_{c_2}}$ is trivially implied by the "until" semantics.

  Let ρ be any timed word, $i \in \mathbb{N}$ and ν be any valuation over free variables of ψ. $\rho, i, \nu \models \psi{\downarrow_{c_1}} \wedge \psi{\downarrow_{c_2}}$, implies that for $j \in \{1, 2\}$, there exists point $i_j > i$ such that $\rho, i_j, \nu \models \beta{\downarrow_{c_j}}$ and all the points between i and i_j satisfies α. It suffices to show that for some $j \in \{1, 2\}$, $\rho, i_j, \nu \models \beta{\downarrow_{c_1}} \wedge \beta{\downarrow_{c_2}}$. Without loss of generality, assume $i_1 \leq i_2$. Then, by Lemma 4.4 and $\rho, i_2, \nu \models \beta{\downarrow_{c_2}}$, we have $\rho, i_1, \nu \models \beta{\downarrow_{c_2}}$, thus $\rho, i_1, \nu \models \beta{\downarrow_{c_1}} \wedge \beta{\downarrow_{c_2}}$. ◀

4.2 Reducing UPTL to finite disjunctions of CoT

Let $\bigvee$ CoT denote the class of UPTL formulas which are finite disjunctions of CoT formulas. In this section, we show that for any UPTL formula φ we can construct an equivalent formula ψ in $\bigvee$ CoT. Hence, CoT satisfies the property **P2**. This is the most technically involved part of the paper; the rest of this section is devoted to the proof of Lemma 4.6.

▶ **Lemma 4.6.** *Any* UPTL *formula $\varphi(x_1, \ldots, x_k)$ such that $\sim \in \mathsf{Pol}(\varphi(x_1, \ldots, x_k))$ can be expressed as a finite disjunction of* CoT *formulas ψ such that $\sim \in \mathsf{Pol}(\psi)$.*

Fix a UPTL formula φ. Let $\bigvee \mathsf{CoT}^{\varphi}$ be the class of UPTL formulas such that $\mathsf{Pol}(\varphi) = \mathsf{Pol}(\psi)$ for all $\psi \in \bigvee$ CoT. By structural induction on φ, we construct an equivalent Θ in $\bigvee \mathsf{CoT}^{\varphi}$.

Base Case. Assume φ is a UPTL formula which is either a closed formula or an atomic time constraint of the form $(T - x) \sim u$. The lemma trivially holds in this case.

Induction Hypothesis. Assume that the result holds for all the strict subformulas of φ.

Induction. We show that φ can be reduced to an equivalent formula in $\bigvee \mathsf{CoT}^{\varphi}$. There are three possible cases. (Case 0) : $\varphi = \alpha \wedge \beta$ or $\varphi = \alpha \vee \beta$, (Case 1) : $\varphi = \alpha \,\mathbf{U}\, \beta$ or $\alpha \,\mathbf{S}\, \beta$, (Case 2) : $\varphi = \Box\alpha$ or $\overleftarrow{\Box}\alpha$. We discuss the more interesting cases 1 and 2. For a full formal proof, see Appendix C. We observe a simplification for cases 1, 2. By induction hypothesis, α and β can be reduced to $\alpha' = \bigvee\limits_{m} \alpha_m$ and $\beta' = \bigvee\limits_{l} \beta_l$ in $\bigvee \mathsf{CoT}^{\varphi}$, respectively. Thus, without loss of generality, we consider cases 1 and 2 to be of the forms $\varphi = \alpha' \,\mathbf{U}\, \beta'$ (or $\alpha' \,\mathbf{S}\, \beta'$) and $\Box\alpha'$ (or $\overleftarrow{\Box}\alpha'$) respectively.

For case 1, we argue about $\mathbf{U}$ formulas (case of $\mathbf{S}$ is symmetric). Notice that $\phi \,\mathbf{U}\, (\psi \vee \psi') \equiv (\phi \,\mathbf{U}\, \psi) \vee (\phi \,\mathbf{U}\, \psi')$. Hence, φ in case 1 can be further simplified by pulling the top-level disjunctions of β'. Hence, $\varphi \equiv \bigvee\limits_{l}(\alpha' \,\mathbf{U}\, \beta_l)$, where each β_l is in CoT having the same polarity as φ. As $\bigvee \mathsf{CoT}^{\varphi}$ is closed under disjunctions, it suffices to show that each $(\alpha' \,\mathbf{U}\, \beta_l)$ is in CoT. This is shown by Lemma 4.7. Moreover φ in case 2 can be reduced to φ in case 1 by Lemma 4.8. The following lemma handles the case 1.

▶ **Lemma 4.7.** *Any* UPTL *formula φ of the form $(\bigvee\limits_{g=1}^{n} \alpha_g) \,\mathbf{U}\, (\beta)$ or $(\bigvee\limits_{g=1}^{n} \alpha_g) \,\mathbf{S}\, (\beta)$, where $\alpha_i, \beta \in \bigvee \mathsf{CoT}^{\varphi}$, can be reduced to an equivalent $\bigvee \mathsf{CoT}^{\varphi}$ formula.*

Proof. We argue for $\mathbf{U}$, and $\mathbf{S}$ is symmetric. We induct on the number of open α_gs in φ. If there are no open α_gs in φ, then φ is, by definition, in $\bigvee \mathsf{CoT}^{\varphi}$.

Induction Hypothesis. Suppose the lemma holds for the case where k of the α_gs are open. We show the lemma holds for formulas having $(k+1)$ open α_gs. Without loss of generality, assume α_1 is open. $\rho, i, \nu \models \varphi$ is equivalent to (A) $\exists j > i$ s.t. $\rho, j, \nu \models \beta$, and $\forall i < j' < j$, $\rho, j', \nu \models \bigvee_{g=1}^{n} \alpha_g$.

Case 1.1. $\leq_{\in} \mathsf{Pol}(\varphi)$:

- Notice that (A) is equivalent to (B1) or (B2), where
  - (B1) $\rho, i, \nu \models (\bigvee\limits_{g=2}^{n} \alpha_g) \,\mathbf{U}\, \beta$ (this is when $\rho, i, \nu \models \neg\alpha_1 \,\mathbf{U}\, \beta$) or
  - (B2) $\exists i' > i$ such that (B2,1)$\rho, i', \nu \models \alpha_1$, (B2,2)$\forall i < i'' < i' : \rho, i'', \nu \models \bigvee\limits_{g=1}^{n} \alpha_g$, (B2,3) $\exists j > i'$ such that $\rho, j, \nu \models \beta$, (B2,4) $\forall i' < j' < j : \rho, j', \nu \models \bigvee_{g=2}^{n} \alpha_g$. Intuitively, after the point i', α_1 need not hold for $\bigvee\limits_{g=1}^{n} \alpha_g$ to hold at point i for a given valuation ν. The last occurrence of α_1 is a valid choice for i'. See Figure 4 in Appendix C.1 for intuition.
- Lemma 4.4 and (B2,1) imply that for all points i'' strictly between i and i', α_1 can be replaced by the closed formula α_1' where $\alpha_1' \in \mathsf{Untime}(\alpha_1)$ is obtained by replacing all open constraints of α_1 with $\top$.
- Hence, the given formula φ is equivalent to $\varphi_1 \vee \varphi_2$ where:
  - $\varphi_1 = [(\bigvee\limits_{g=2}^{n} \alpha_g) \,\mathbf{U}\, \beta]$, $\varphi_2 = [(\bigvee\limits_{g=2}^{n} \alpha_g \vee \alpha_1') \,\mathbf{U}\, (\alpha_1 \wedge \varphi_1)]$
- By induction hypothesis, φ_1 can be reduced to a formula of the form $\bigvee\limits_{h=1}^{m} (\Theta_{1,h})$ where each $\Theta_{1,h}$ is in $\bigvee \mathsf{CoT}^{\varphi}$ having the same polarity as φ. Substituting this for φ_1 in φ_2 and pushing all the disjunctions at the top, we get $\Theta_2 = \bigvee\limits_{h=1}^{m} [(\bigvee\limits_{g=2}^{n} \alpha_g \vee \alpha_1') \,\mathbf{U}\, (\alpha_1 \wedge \Theta_{1,h})]$
- It suffices to show that each $\Theta_{2,h} = [(\bigvee\limits_{g=2}^{n} \alpha_g \vee \alpha_1') \,\mathbf{U}\, (\alpha_1 \wedge \Theta_{1,h})]$ can be reduced to an equivalent $\bigvee \mathsf{CoT}^{\varphi}$ as formulas in $\bigvee \mathsf{CoT}^{\varphi}$ are closed under disjunctions. Moreover, $\Theta_{2,h}$

has k open α_i formulas as one of the open formula α_1 is replaced with a closed formula α_1'. Hence, by induction hypothesis, this could be reduced to an equivalent formula in $\bigvee \mathsf{CoT}^{\varphi}$, proving the result.

Case 1.2. $\geq \in \mathsf{Pol}(\varphi)$: We discuss this case in Appendix C.1. ◀

Case 2 is handled by the following Lemma 4.8.

▶ **Lemma 4.8.** *Let $\varphi \in \mathsf{UPTL}$ be of the form $\Box(\bigvee_{g=1}^{n} \alpha_g)$ or $\overleftarrow{\Box}(\bigvee_{g=1}^{n} \alpha_g)$, where each $\alpha_g \in \mathsf{CoT}$. Then, we can construct a $\bigvee \mathsf{CoT}^{\varphi}$ formula equivalent to φ.*

Notice that $\overleftarrow{\Box}(\bigvee_{g=1}^{n} \alpha_g) \equiv (\bigvee_{g=1}^{n} \alpha_g)\, \mathbf{S}\, (start)$, where $start = \overleftarrow{\Box} \bot$, is true only at the first point of the timed word. Hence, now applying Lemma 4.7, we get the required result.

Let us now consider $\varphi = \Box(\bigvee_{g=1}^{n} \alpha_g)$. There are two possible subcases:

- **Case 2.1** $\leq \in \mathsf{Pol}(\varphi)$, and **Case 2.2** $\geq \in \mathsf{Pol}(\varphi)$. The latter can be reduced to a formula in Case 1.2, and the proof appears in the Appendix D.
- **Case 2.1** $\leq \in \mathsf{Pol}(\varphi)$. This is the most interesting of all the cases. The idea is to reduce this to Case 1.1 of Lemma 4.7 wherever possible. Let $\alpha' = \alpha_1 \vee \ldots \vee \alpha_n$, and $\varphi = \Box(\alpha')$. If we restrict to finite timed words then this problem is trivially reduced to Case 1.1 of Lemma 4.7 as $\Box(\alpha') \equiv \alpha'\, \mathbf{U}\, Last$, where $Last = \Box(\bot)$ is the formula which evaluates to true only at the last point. The infinite word case needs more machinery.

4.3 Machinery for Case 2.1, Lemma 4.8 for Infinite Words

We define some terminology which would be useful in the rest of the proof.

▶ **Definition 4.9** (Finitely Occurring Formulas). *Given $\psi(x_1, \ldots, x_k) \in \mathsf{TPTL}$, a timed word ρ, and a valuation for free variables of ψ, we say that ψ is finitely occurring in ρ, ν iff there are only finitely many points i of ρ such that $\rho, i, \nu \models \psi$. That is, there are only finitely many points in ρ where ψ is satisfied for a given valuation.*

▶ **Definition 4.10** (Maximal Finitely Occurring Open Subformula (MFOS)). *Given any* TPTL *formula $\varphi(x_1, \ldots, x_k)$, a timed word ρ, and a valuation ν for free variables of φ, we say that a subformula ψ of φ is a "finitely occurring open subformula" (FOS) of φ in ρ, ν iff*

1. *ψ is not a subformula of any closed subformula ψ' of φ. This implies ψ is open.*
2. *ψ is finitely occurring in ρ, ν.*

ψ is said to be a "maximal finitely occurring open subformula" (MFOS) of φ in ρ, ν iff ψ is a maximal formula that satisfies the above two conditions: that is, ψ is not a strict subformula of an open, finitely occurring formula in ρ, ν, and ψ is not a subformula of a closed formula.

▶ **Proposition 4.11.** *For any timed word ρ and a valuation ν, formulas of the form $(T - x) \leq u$ (or $(T - x) < u$) are finitely occurring formulas in ρ, ν.*

Proof. This is trivial, as otherwise this would imply that there are infinitely many points within at most u distance apart, implying the Zeno condition. ◀

▶ Remark 4.12. Any formula of φ that has a FOS also has a MFOS. Hence, by Proposition 4.11, any TPTL formula φ that contains an open constraint of the form $(T - x) \leq u$ or $(T - x) < u$ has an MFOS.

▶ **Proposition 4.13.** *Given* TPTL *formulas φ, ψ, a timed word ρ and a valuation ν over the free variables of φ and ψ, if ψ is finitely occurring in ρ, ν, then both $\varphi \,\mathbf{U}\, \psi$, and $\varphi \wedge \psi$ are finitely occurring.*

Proof. If ψ holds at finitely many points of ρ for the given valuation, then there exists a point j in ρ after which ψ ceases to hold for the given valuation ν. By semantics of $\mathbf{U}$ and conjunction, any point after j can not satisfy $\varphi \,\mathbf{U}\, \psi$ and $\varphi \wedge \psi$ for the given valuation ν. Hence, these are also finitely occurring in ρ, ν. ◀

▶ **Proposition 4.14.** *Given any* CoT *formulas φ, ψ, a timed word ρ, and a valuation ν over the free variables of φ and ψ, if ψ is MFOS of φ in ρ, ν, then either $\varphi = \psi$ or ψ appears in a subformula $\beta \,\mathbf{S}\, \psi$ of φ. In other words, either ψ is the same as φ or ψ appears as the right argument of a $\mathbf{S}$ modality. Moreover, this $\beta \,\mathbf{S}\, \psi$ is not finitely occurring.*

Proof. Suppose this is not the case. If ψ is in the scope of $\vee$, $\Box$, or $\overleftarrow{\Box}$ then φ is not CoT as ψ is an open subformula. Then ψ is in the scope of $\mathbf{U}$, $\mathbf{S}$ or $\wedge$: φ has a subformula of the form $\beta \,\mathbf{U}\, \psi$, $\beta \wedge \psi$ or $\beta \,\mathbf{S}\, \psi$. For the first and second cases, as ψ is finitely occurring in ρ, ν, by Proposition 4.13, both $\beta \,\mathbf{U}\, \psi$ and $\beta \wedge \psi$ are finitely occurring in ρ, ν. This implies ψ is not a MFOS of φ in ρ, ν, a contradiction. Thus, the only option is $\beta \,\mathbf{S}\, \psi$. If $\beta \,\mathbf{S}\, \psi$ is finitely occurring in ρ, ν, then ψ is not MFOS of φ in ρ, ν. Hence, $\beta \,\mathbf{S}\, \psi$ is not finitely occurring. ◀

▶ **Proposition 4.15.** *Given* TPTL *formulas φ, ψ, a timed word ρ, and a valuation ν over the free variables of φ and ψ, if ψ is finitely occurring for ρ, ν then for any point i of ρ,*

I. $\rho, i, \nu \models \Diamond\Box(\varphi \,\mathbf{S}\, \psi) \iff \rho, i, \nu \models \Box\Diamond(\varphi \,\mathbf{S}\, \psi)$, *and*

II. $\rho, i, \nu \models \Box(\varphi \,\mathbf{S}\, \psi) \iff \rho, i, \nu \models \chi$ *where* $\chi = \chi^+ \vee \chi^- \vee \chi^0$ *and*
$\chi^+ = (\varphi \,\mathbf{S}\, \psi) \,\mathbf{U}\, (\psi \wedge \Box(\varphi)), \quad \chi^- = (\varphi \,\mathbf{S}\, \psi) \,\mathbf{S}\, (\psi \wedge \Box(\varphi)), \quad \chi^0 = \psi \wedge \Box(\varphi).$

Proof. Let i be any point of ρ. As ψ is finitely occurring, there exists a point j such that (a) $\rho, j, \nu \models \psi$, and $\rho, j', \nu \not\models \psi$ for any $j' > j$.

Proving I. The forward direction is trivially a tautology. For the reverse direction, suppose j satisfies $\rho, j, \nu \models \Box\Diamond(\varphi \,\mathbf{S}\, \psi)$. That is, there are infinitely many points after j satisfying $\varphi \mathbf{S} \psi$. Hence, for any point $i > j$, we can find a point $j'' > i > j$ such that (b) $\rho, j'', \nu \models \varphi \mathbf{S} \psi$. (see Figure 3 along with the proof for intuition).

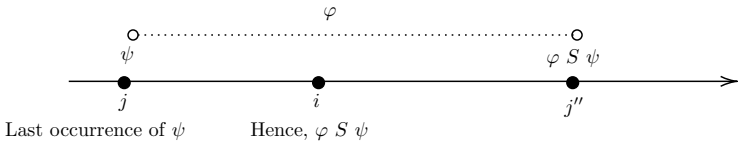


Figure 3 Figure depicting intuition behind Proposition 4.15 (I).

But, by (a), the closest point where ψ holds before j'' is j. Hence, (a) and (b) imply $\rho, i, \nu \models \varphi$. Hence, every point after i necessarily satisfies φ and hence $\varphi \,\mathbf{S}\, \psi$. Thus, every point after j also satisfies $\varphi \,\mathbf{S}\, \psi$. Hence, (c) : for any point i in ρ, there is a point k in the future of i, such that all the points $k' \geq k$ satisfy $\varphi \,\mathbf{S}\, \psi$ for the valuation ν. Notice that $k = i$ in case $i > j$ and $k = j + 1$ in case $i \leq j$. (c) shows the reverse direction of I.

Proving II. In case of II, the reverse direction is a tautology (by semantics). For the forward direction: let (Assumption 1) $\rho, i, \nu \models \Box(\varphi \,\mathbf{S}\, \psi)$ and (Assumption 2) ψ be finitely occurring in ρ, ν. By I, (Assumption 1) and (Assumption 2), we know that there exists a point i' such that $\rho, i', \nu \models \chi^0$, where $\chi^0 = \psi \wedge \Box(\varphi)$. One such point is j, the last occurrence of ψ (see (c) in the proof of I). If $i < i'$, then $\chi^+ = (\varphi \,\mathbf{S}\, \psi) \,\mathbf{U}\, (\psi \wedge \Box(\varphi))$ holds at i. For $i > i'$, we encode the fact that i comes after i', by asserting $\chi^- = \overleftarrow{\Diamond}((\psi \wedge \Box(\varphi))$ at i. For $i = j$, we encode that j is same as i : $\chi^0 = \psi \wedge \Box(\varphi)$. ◀

4.3.1 Case 2.1: Handling Infinite Words

Recall that $\leq \in \mathsf{Pol}(\varphi)$ and $\varphi = \Box\alpha'$ with $\alpha' = \alpha_1 \vee \cdots \vee \alpha_n$. We apply induction on the formula size of α'. If the size of α' is 1, the construction is trivial. We either get a closed formula of the form $\Box((T-x) \leq c)$ or $\Box((T-x) < c)$. Both these evaluate to $\bot$ (which is a closed formula) as they imply zenoness (see Proposition 4.11), hence getting a formula in CoT.

- If all α_i are closed then φ is closed, and hence trivially in CoT. Hence, we consider the case where at least one of the α_i is open. Wlg, assume α_1 is open. We now show that we can construct an equivalent formula for $\varphi = \Box(\alpha')$ in $\bigvee \mathsf{CoT}^{\varphi}$, assuming such a construction exists for all formulas smaller than φ.

  Let ρ be any timed word, i any point in ρ, and ν any valuation over free variables of φ. When we mention about the satisfaction of a TPTL formula at any point in ρ, we assume it is for the given valuation ν. There are three possible cases. We only discuss the case (called Case C below) when α_1 is not finitely occurring in ρ, ν, which mandates the use of MFOS. Appendix D.1 has the other two (called cases A and B) and are similar to the finite word case.
- Case C : α_1 *is not finitely occurring in* ρ, ν. Let i be any point of ρ. Let δ be any subformula of α_1 such that **[I]** $\rho, i, \nu \models \Diamond\Box(\delta)$. That is, $\exists j > i$ such that $\rho, j, \nu \models \Box(\delta)$. Hence, after point j we can replace the check of α_1 with α_{1_δ} where α_{1_δ} is obtained from α_1 by replacing δ to $\top$. Thus, $\rho, i, \nu \models \varphi$ iff $\rho, i, \nu \models \varphi_{3,\delta}$ where
  $\varphi_{3,\delta} = \alpha' \,\mathbf{U}\, [\alpha' \wedge \Box(\delta) \wedge \varphi_\delta]$, and $\varphi_\delta = \Box(\alpha_{1_\delta} \vee \alpha_2 \vee \ldots \vee \alpha_n)$.
  Moreover, by TPTL/LTL semantics, we get **[II]** if $\rho, i, \nu \not\models \Diamond\Box(\delta)$, then $\varphi_{3,\delta}$ is equivalent to false. That is, $\rho, i, \nu \not\models \varphi_{3,\delta}$. This is because $\varphi_{3,\delta}$ implies $\rho, i, \nu \models \Diamond\Box(\delta)$. Let $SFS(\alpha_1)$ be all the open subformulas of α_1 with $\mathbf{S}$ being the top level modality, i.e. formulas of the form $\delta = \beta \,\mathbf{S}\, \gamma$. We now show that : $\varphi \equiv \bigvee_{\delta \in SFS(\alpha_1)} \varphi_{3,\delta}$.

  For that, it suffices to show **[III]** Case C implies there is at least one $\delta \in SFS(\alpha_1)$ such that $\rho, i, \nu \models \Diamond(\Box(\delta))$ for any $i \in dom(\rho)$. **[II]** and **[III]** imply for any $i \in dom(\rho)$ there is at least one disjunct equivalent to φ, and every disjunct is either equivalent to $\bot$ or φ. We now prove **[III]**. As α_1 is not a closed formula, it has at least one MFOS with respect to ρ, ν (Remark 4.12). By Proposition 4.14, any MFOS of α_1 with respect to ρ, ν should occur within the scope of temporal operator $\mathbf{S}$; hence, α_1 has a subformula of the form $\delta = \beta \,\mathbf{S}\, \gamma$ where γ is an MFOS with respect to ρ, ν. In other words, there is at least one formula $\beta \,\mathbf{S}\, \gamma \in SFS(\alpha_1)$. Moreover, as γ is "maximal" finitely occurring subformulas, $\delta = \beta \mathbf{S} \gamma$ is not finitely occurring. This along with Proposition 4.15 imply $\rho, i, \nu \models \Diamond\Box(\delta)$, hence proving **[III]**. Thus $\varphi \equiv \bigvee_{\delta \in SFS(\alpha_1)} \varphi_{3,\delta}$. As $\bigvee \mathsf{CoT}^{\varphi}$is closed under disjunctions among the same polarity formulas, we just need to show that $\varphi_{3,\delta}$ for any δ is in $\bigvee \mathsf{CoT}^{\varphi}$, which is what we show now. Since φ' is strictly smaller than φ, by induction hypothesis, there is a formula ψ'_δ in $\bigvee \mathsf{CoT}^{\varphi}$equivalent to φ'. Moreover, by Proposition 4.15(II), there is $\chi = \chi^+ \vee \chi^- \vee \chi^0$ equivalent to $\Box(\beta \,\mathbf{S}\, \gamma)$. χ^0 is already in CoT with the same polarity as φ. χ^+ and χ^- are $\mathbf{U}$ and $\mathbf{S}$ formulas not necessarily in $\bigvee \mathsf{CoT}^{\varphi}$. But, it reduces to the **Case 1.1**. Thus, using the construction in **Case 1.1**, we can construct an equivalent formula χ' in $\bigvee \mathsf{CoT}^{\varphi}$ which is equivalent to χ. Hence, we get the following formula equivalent to $\varphi_{3,\delta}$, namely, $\varphi'_{3,\delta} = \alpha' \,\mathbf{U}\, [\alpha' \wedge \chi' \wedge \psi'_\delta]$. This again falls under **Case 1.1** of Lemma 4.7. Hence, we can construct a ψ_δ, for a given δ which is in $\bigvee \mathsf{CoT}^{\varphi}$.
- The formula we are looking for is obtained by disjuncting the formulas constructed in all three cases A, B, C (of which we have Case C here, and Cases A,B in Appendix D.1).

Lemmas 4.5 and 4.6 imply Lemma 3.6. For any UPTL formula $\psi(x_1, \ldots, x_k)$, construct an equivalent $\psi'(x_1, \ldots, x_k) = \bigvee_{i=1}^{m} \psi_i(x_1, \ldots, x_k)$, where each ψ_i is in CoT with the same polarity as that of ψ. Then apply Lemma 4.5 and reduce each $\psi_i(x_1, \ldots, x_k)$ to $\bigwedge_{j=1}^{n} \psi_{i_j}(x_{i,j})$, where each $x_{i,j}$ is some variable in $\{x_1, \ldots, x_k\}$, and ψ_{i_j} contains at most one open time constraint. Then ψ is equivalent to $\varphi = \bigvee_{i=1}^{m} \bigwedge_{j=1}^{n} \psi_{i_j}(x_i)$, obtaining the result. ◀

▶ Remark 4.16. Each ψ_{i_j} is a CoT containing at most one open constraint. Thus, our construction gives a stronger result than Lemma 3.6, which is used in proving Theorem 3.5.

▶ **Lemma 4.17** (Monadic Decomposition of UPTL (Stronger Form)). *There is an effective translation from any* UPTL $\varphi(x_1, \ldots, x_k)$ *to an equivalent* UPTL $\psi = (\Psi_1) \vee (\Psi_2) \vee \ldots \vee (\Psi_n)$ *where each* Ψ_i *is a conjunction of* CoT *formulas containing at most one open time constraint.*

5 1-UPTL to Q2MLO: Proof of Theorem 3.5

By Theorem 3.3, Q2MLO is equivalent to UMITPPL. Moreover, by Remark 2.1, 1-UPTL can efficiently express formulas in UMITPPL. Hence, to prove Theorem 3.5, it suffices to show that we can convert a closed 1-UPTL formula φ to an equivalent Q2MLO formula $\psi(y)$. We give a proof sketch along with a running example.

1. Let $\varphi{=}x.\varphi'(x)$ be a closed 1-UPTL formula. Using Lemma 4.17 on φ', and distributing $(x.)$ over disjunctions, we get an equivalent 1-UPTL formula $\varphi_{DTF} = [(\varphi_1) \vee (\varphi_2) \vee \ldots \vee (\varphi_k)]$, where each φ_i has the form $x.\varphi_{1,i} \wedge \ldots \wedge x.\varphi_{k,i}$ and each $\varphi_{j,i}$ is a formula in CoT containing at most one open time constraint. As Q2MLO is closed under $\vee$s and $\wedge$s, it suffices to show that each $x.\varphi_{j,i}$ is reducible to a Q2MLO formula $\psi'(y)$.

   Let $\varphi'(x) = a \,\mathbf{U}\, [\{((T{-}x) {<} u_1) \wedge (a \,\mathbf{S}\, (b \wedge ((T{-}x) {<} u_2)))\} \vee \{(T{-}x){<}u_3\}]$. The above reduction on φ' gives $\varphi_{DTF} = [\varphi_{1,1} \wedge \varphi_{1,2}] \vee [\varphi_{2,1}]$ where, $\varphi_{2,1} = a \,\mathbf{U}\, \{(T{-}x){<}u_3\}$, $\varphi_{1,1} = a \,\mathbf{U}\, \{((T{-}x) {<} u_1) \wedge (a \,\mathbf{S}\, b))\}$, $\varphi_{1,2} = a \,\mathbf{U}\, \{(a \,\mathbf{S}\, (b \wedge ((T{-}x) {<} u_2)))\}$.
2. Encode the semantics of $\varphi_{j,i}$ in first order logic. Notice that the single open constraint in a CoT formula $\varphi_{j,i}$ never appears on a left subtree of any $\mathbf{U}$ or $\mathbf{S}$ modality in the parse tree of $\varphi_{j,i}$. Thanks to this syntactic restriction of CoT, the variable guarded by metric predicates always appears within the scope of $\exists^*$ in the corresponding first-order formula. After encoding the semantics in first-order logic and pulling up the "metric guarded quantifiers", i.e. the quantifier $\exists y$ quantifying a variable appearing in the open timing constraint $y \ominus x \sim c$, to the top gives us an equivalent Q2MLO formula.

   To illustrate, we show the reduction on $x.\varphi_{1,2}$, where the open time constraint appears at modal depth two. Similar reductions can be applied for $x.\varphi_{1,1}$ and $x.\varphi_{2,1}$. By TPTL semantics, $\rho, j \models x.\varphi_{1,2}$ iff $\rho, \nu \models \psi_{1,2}(x)$ where $\psi_{1,2}(x)$ is the required Q2MLO defined by $\nu(x) = j$ and
   $\psi_{1,2}(x){=}\exists z.[(z \ominus x < u_2) \wedge \varphi(x, z)]$ where $\varphi(x, z)$ is defined as follows.
   $\varphi(x, z) = \exists y.[\{z{<}y \wedge x < y \wedge b(z)\} \wedge \{(\forall y'.(x{<}y'{<}y{\rightarrow}a(y')))\wedge\{\forall z'(z{<}z'{<}y{\rightarrow}a(z'))\}]$.

6 Applications

Theoretical Applications. We first discuss the theoretical implications of our results. First, our results resolve the open questions of [24] : (1) Does the expressiveness of UPTL[Fut] increase with more variables? We resolve this question negatively. Notice that the variable reduction in the proof of Theorem 3.4 also works for the future fragment of UPTL. Thus,

▶ **Corollary 6.1.** UPTL[Fut] $\equiv$ 1-UPTL[Fut]

(2) Is full UPTL decidable? We resolve this question positively. Theorem 3.1 and the decidability of the satisfiability checking problem of Q2MLO [11] implies:

▶ **Corollary 6.2.** *Satisfiability checking for* UPTL *is decidable.*

Practical Applications: Compositional Verification of UPTL. A fundamental challenge in verifying freeze logics like UPTL is the lack of compositional techniques due to the dependency of subformulas on multiple free variable valuations. Our reduction addresses this by transforming UPTL formulas into equivalent MITPPL formulas, enabling compositional verification via model checkers such as MightyL [7], Tchecker [10] (both accept MITL[Fut] formulas) and MightyPPL [27] (accepts MITPPL formulas). Our reduction allows us to leverage small, modular timed automata representations, significantly improving scalability. Consider a food delivery service specification as follows. The food delivery should be quick (within 30 mins since the order) and fresh (within 15 mins since food is cooked/pick-up). The UPTL formula $\psi = \Box(\text{order} \Rightarrow \phi)$ over propositions $AP = \{\text{order, pickup, deliver}\}$ where $\phi = x.\Diamond(\text{pickup} \wedge y.\Diamond(\text{deliver} \wedge x < 30 \wedge y < 15))$ captures this. Our technique translates ϕ to the equivalent MITPPL formula $\phi' = \mathbf{Pn}_{[0,30]}(\text{pickup,deliver}) \wedge \Diamond(\text{pickup} \wedge \Diamond_{[0,15]}(\text{deliver}))$. MightyPPL took 10ms to construct the timed automaton corresponding to ϕ' and 1ms to verify that it is satisfiable.

7 Conclusion and Discussion

One of our key contributions is a *first equivalence result* between a *freeze logic* and a *metric logic* under *pointwise semantics*. Our results extend naturally to the *continuous semantics*. This equivalence serves as an analogue to known results in continuous semantics, such as the equivalence between MTL *with counting and* TPTL [18], and between MTL[$\mathbb{Q}$] *and* TPTL[$\mathbb{Q}$] [19]. These prior equivalences break down in *pointwise semantics*. An interesting consequence of our results is that adding variables does not increase the expressiveness of UPTL, while introducing past operators does (See Appendix E).

(1) Precise Complexities, Translation Sizes. We conjecture that the satisfiability and model checking of UPTL to be in EXPSPACE, based on the observation that many subformulas in the reduced 1-UPTL formula are repeated. Hence, one could hope that the upper bound on the *formula DAG* of the equivalent 1-UPTL could be only *exponential* in the size of the input UPTL formula. Additionally, if the given UPTL is already in the $\bigvee$CoT form, then the UPTL to 1-UPTL reduction is worst case quadratic (Lemma 4.5). Translating 1-UPTL to UMITPPL can get non-elementary (as Q2MLO generalizes FO[$<$] over words, and has a non-elementary satisfiability). While our focus is expressive equivalence, for efficient satisfiability checking, 1-UPTL can be reduced to equisatisfiable UMITPPL with only exponential blowup using ideas from [21]. Translating UPTL to $\bigvee$ CoT causes a non-elementary blowup on formula trees, but using DAGs and a Tseitin-like encoding, we conjecture an exponential equisatisfiability reduction. Our translations, along with tools like MightyPPL [27] can verify UPTL specifications.

Succinctness of UPTL vs. 1-UPTL. While we establish that increasing the number of variables does not improve *expressiveness*, we believe it may lead to an exponential increase in *succinctness*. We leave a formal proof of UPTL (or UPTL[Fut]) being more succinct than 1-UPTL (resp. 1-UPTL[Fut]) as future work.

References

1 Rajeev Alur, Tomás Feder, and Thomas A. Henzinger. The benefits of relaxing punctuality. In Luigi Logrippo, editor, *Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing, Montreal, Quebec, Canada, August 19-21, 1991*, pages 139–152. ACM, 1991. doi:10.1145/112600.112613.

2 Rajeev Alur, Tomás Feder, and Thomas A. Henzinger. The benefits of relaxing punctuality. *J.ACM*, 43(1):116–146, 1996. doi:10.1145/227595.227602.

3 Rajeev Alur and Thomas A. Henzinger. A really temporal logic. In *30th Annual Symposium on Foundations of Computer Science*, pages 164–169, 1989. doi:10.1109/SFCS.1989.63473.

4 Rajeev Alur and Thomas A. Henzinger. Real-time logics: Complexity and expressiveness. *Inf. Comput.*, 104(1):35–77, 1993. doi:10.1006/inco.1993.1025.

5 Patricia Bouyer, Fabrice Chevalier, and Nicolas Markey. On the expressiveness of TPTL and MTL. In Sundar Sarukkai and Sandeep Sen, editors, *FSTTCS 2005: Foundations of Software Technology and Theoretical Computer Science*, pages 432–443, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. doi:10.1007/11590156_35.

6 Thomas Brihaye, Morgane Estiévenart, and Gilles Geeraerts. On mitl and alternating timed automata over infinite words. In *Formal Modeling and Analysis of Timed Systems: 12th International Conference, FORMATS 2014, Florence, Italy, September 8-10, 2014. Proceedings 12*, pages 69–84. Springer, 2014. doi:10.1007/978-3-319-10512-3_6.

7 Thomas Brihaye, Gilles Geeraerts, Hsi-Ming Ho, and Benjamin Monmege. Mightyl: A compositional translation from MITL to timed automata. In Rupak Majumdar and Viktor Kuncak, editors, *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*, volume 10426 of *Lecture Notes in Computer Science*, pages 421–440. Springer, 2017. doi:10.1007/978-3-319-63387-9_21.

8 Thomas Ferrère. The compound interest in relaxing punctuality. In Klaus Havelund, Jan Peleska, Bill Roscoe, and Erik P. de Vink, editors, *Formal Methods - 22nd International Symposium, FM 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 15-17, 2018, Proceedings*, volume 10951 of *Lecture Notes in Computer Science*, pages 147–164. Springer, 2018. doi:10.1007/978-3-319-95582-7_9.

9 Thomas A. Henzinger, Jean-François Raskin, and Pierre-Yves Schobbens. The regular real-time languages. In Kim Guldstrand Larsen, Sven Skyum, and Glynn Winskel, editors, *Automata, Languages and Programming, 25th International Colloquium, ICALP'98, Aalborg, Denmark, July 13-17, 1998, Proceedings*, volume 1443 of *Lecture Notes in Computer Science*, pages 580–591. Springer, 1998. doi:10.1007/BFb0055086.

10 Frédéric Herbreteau and Gérald Point. The tchecker tool and libraries. https://github.com/ticktac-project/tchecker.

11 Yoram Hirshfeld and Alexander Moshe Rabinovich. A framework for decidable metrical logics. In Jirí Wiedermann, Peter van Emde Boas, and Mogens Nielsen, editors, *Automata, Languages and Programming, 26th International Colloquium, ICALP'99, Prague, Czech Republic, July 11-15, 1999, Proceedings*, volume 1644 of *Lecture Notes in Computer Science*, pages 422–432. Springer, 1999. doi:10.1007/3-540-48523-6_39.

12 Yoram Hirshfeld and Alexander Moshe Rabinovich. Logics for real time: Decidability and complexity. *Fundamenta Informaticae*, 62(1):1–28, 2004. URL: http://content.iospress.com/articles/fundamenta-informaticae/fi62-1-02.

13 Yoram Hirshfeld and Alexander Moshe Rabinovich. An expressive temporal logic for real time. In *MFCS*, pages 492–504, 2006. doi:10.1007/11821069_43.

14 Yoram Hirshfeld and Alexander Moshe Rabinovich. Expressiveness of metric modalities for continuous time. In Dima Grigoriev, John Harrison, and Edward A. Hirsch, editors, *Computer Science – Theory and Applications*, pages 211–220, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. doi:10.1007/11753728_23.

15 Hsi-Ming Ho. Revisiting timed logics with automata modalities. In Necmiye Ozay and Pavithra Prabhakar, editors, *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control, HSCC 2019, Montreal, QC, Canada, April 16-18, 2019*, pages 67–76. ACM, 2019. doi:10.1145/3302504.3311818.

16 Hsi-Ming Ho and Khushraj Madnani. When do you start counting? revisiting counting and pnueli modalities in timed logics. In *DCM*. EPTCS, 2023.

17 Hsi-Ming Ho and Khushraj Madnani. Metric quantifiers and counting in timed logics and automata. *Information and Computation*, 303:105268, 2025. doi:10.1016/j.ic.2025.105268.

18 Paul Hunter. When is metric temporal logic expressively complete? In *22nd EACSL Conference on Computer Science Logic, CSL*, pages 380–394, 2013. doi:10.4230/LIPICS.CSL.2013.380.

19 Paul Hunter, Joël Ouaknine, and James Worrell. Expressive completeness for metric temporal logic. In *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28, 2013*, pages 349–357. IEEE Computer Society, 2013. doi:10.1109/LICS.2013.41.

20 Ron Koymans. Specifying real-time properties with metric temporal logic. *Real Time Syst.*, 2(4):255–299, 1990. doi:10.1007/BF01995674.

21 Shankara Narayanan Krishna, Khushraj Madnani, Manuel Mazo Jr., and Paritosh K. Pandya. Generalizing non-punctuality for timed temporal logic with freeze quantifiers. In Marieke Huisman, Corina S. Pasareanu, and Naijun Zhan, editors, *Formal Methods - 24th International Symposium, FM 2021, Virtual Event, November 20-26, 2021, Proceedings*, volume 13047 of *Lecture Notes in Computer Science*, pages 182–199. Springer, 2021. doi:10.1007/978-3-030-90870-6_10.

22 Shankara Narayanan Krishna, Khushraj Madnani, and Paritosh K. Pandya. Metric temporal logic with counting. In *Foundations of Software Science and Computation Structures - 19th International Conference, FOSSACS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, pages 335–352, 2016. doi:10.1007/978-3-662-49630-5_20.

23 Shankara Narayanan Krishna, Khushraj Madnani, and Paritosh K. Pandya. Making metric temporal logic rational. In Kim G. Larsen, Hans L. Bodlaender, and Jean-François Raskin, editors, *42nd International Symposium on Mathematical Foundations of Computer Science, MFCS 2017, August 21-25, 2017 - Aalborg, Denmark*, volume 83 of *LIPIcs*, pages 77:1–77:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.MFCS.2017.77.

24 Shankara Narayanan Krishna, Khushraj Nanik Madnani, Rupak Majumdar, and Paritosh K. Pandya. Satisfiability checking of multi-variable TPTL with unilateral intervals is pspace-complete. In Guillermo A. Pérez and Jean-François Raskin, editors, *34th International Conference on Concurrency Theory, CONCUR 2023, September 18-23, 2023, Antwerp, Belgium*, volume 279 of *LIPIcs*, pages 23:1–23:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICS.CONCUR.2023.23.

25 Slawomir Lasota and Igor Walukiewicz. Alternating timed automata. *ACM Trans. Comput. Log.*, 9(2):10:1–10:27, 2008. doi:10.1145/1342991.1342994.

26 Nicolas Markey. Temporal logic with past is exponentially more succinct. *Bulletin-European Association for Theoretical Computer Science*, 79:122–128, 2003.

27 MightyPPL. https://github.com/hsimho/MightyPPL.

28 Joël Ouaknine and James Worrell. On the decidability of metric temporal logic. In *LICS*, pages 188–197, 2005. doi:10.1109/LICS.2005.33.

29 Paritosh K. Pandya and Simoni S. Shah. On expressive powers of timed logics: Comparing boundedness, non-punctuality, and deterministic freezing. In Joost-Pieter Katoen and Barbara König, editors, *CONCUR 2011 - Concurrency Theory - 22nd International Conference, CONCUR 2011, Aachen, Germany, September 6-9, 2011. Proceedings*, volume 6901 of *Lecture Notes in Computer Science*, pages 60–75. Springer, 2011. doi:10.1007/978-3-642-23217-6_5.

30 Jean Francois Raskin. *Logics, Automata and Classical Theories for Deciding Real Time*. PhD thesis, Universite de Namur, 1999.
31 Howard Straubing. *Finite Automata, Formal Logic and Circuit Complexity*. Birkhauser, 1994.
32 Thomas Wilke. Specifying timed state sequences in powerful decidable logics and timed automata. In *Formal Techniques in Real-Time and Fault-Tolerant Systems, Third International Symposium Organized Jointly with the Working Group Provably Correct Systems - ProCoS, Lübeck, Germany, September 19-23, Proceedings*, pages 694–715, 1994. doi:10.1007/3-540-58468-4_191.

A Proof of Lemma 4.4

Proof. Let ψ be a formula in CoT. We prove for the case $\leq \in \mathsf{Pol}(\psi)$; a symmetric argument follows when $\geq \in \mathsf{Pol}(\psi)$. Fix a timed word $\rho = (a_1, \tau_1)(a_2, \tau_2)\ldots$, a point j of ρ and a valuation ν over the variables appearing in ψ. Let $\psi' \in \mathsf{Untime}(\psi)$. Assuming that (A) $\rho, j, \nu \models \psi$, we now show that (B) for any point $i \leq j$, $\rho, i, \nu \models \psi$ iff $\rho, i, \nu \models \psi'$. We apply induction on the structure of the formula ψ. From now, we assume that (A) holds.

Base Case. Let ψ be either a closed formula or an atomic clock constraint of the form $T - x \sim u$ where $\sim \in \{<, \leq\}$. If ψ is a closed formula then $\mathsf{Untime}(\psi) = \{\psi\}$. Hence, the lemma holds trivially. If $\psi = (T - x) \sim u$, then $\mathsf{Untime}(\psi) = \{\psi, \top\}$. It suffices to show that (I) $\rho, i, \nu \models \top$ iff (II) $\rho, i, \nu \models (T - x) \sim u$ for all $i \leq j$. (II) implies (I) is a tautology. Conversely, for any $i \leq j$, $\tau_i \leq \tau_j$. As (A) holds, we know $\tau_j - \nu(x) \leq u$; hence for any $i \leq j$, $\tau_i - \nu(x) \leq u$, giving $\rho, i, \nu \models (T - x) \leq u$.

Induction Hypothesis. Assume the lemma holds for all subformulas of ψ.

Induction. We now show that (B) holds given the induction hypothesis and (A). There are three possible cases.

1. $\psi = \psi_1 \wedge \psi_2$. By definition of Untime, $\psi' = \psi'_1 \wedge \psi'_2$ such that $\psi'_1 \in \mathsf{Untime}(\psi_1)$ and $\psi'_2 \in \mathsf{Untime}(\psi_2)$. Now, (A) iff (A1.1) $\rho, j, \nu \models \psi_1$ and (A1.2) $\rho, j, \nu \models \psi_2$. As ψ is a formula in CoT, both ψ_1 and ψ_2 are in CoT. Hence, by induction hypothesis, (A1.1) and (A1.2) implies that for all $i \leq j$, (B1.1) $\rho, i, \nu \models \psi_1$ iff $\rho, i, \nu \models \psi'_1$ and (B1.2) $\rho, i, \nu \models \psi_2$ iff $\rho, i, \nu \models \psi'_2$, respectively. Thus, (A) implies (B). Notice that this argument may not hold when $\psi = \psi_1 \vee \psi_2$. But in this case, ψ is not a CoT formula as disjunction in CoT are allowed only amongst closed formulas.
2. $\psi = \psi_1 \mathbf{U} \psi_2$. As ψ is in CoT, ψ_1 is a closed formula. Hence, $\mathsf{Untime}(\psi_1) = \{\psi_1\}$, and $\psi' = \psi_1 \mathbf{U} \psi'_2$ where $\psi'_2 \in \mathsf{Untime}(\psi_2)$. We show that assuming (A), for any $i \leq j$, (B2.1) $\rho, i, \nu \models \psi$ iff (B2.2) $\rho, i, \nu \models \psi'$. By semantics of $\mathbf{U}$,
   A holds iff there exists a point j' such that (I) $j' > j$, $\rho, j', \nu \models \psi_2$, and for any point j'' between j and j', $\rho, j'', \nu \models \psi_1$.
   B2.1 holds iff there exists a point i' such that (II) $i' > i$, $\rho, i', \nu \models \psi_2$, and for any point i'' between i and i', $\rho, i'', \nu \models \psi_1$.
   B2.2 holds iff there exists a point i' such that (III) $i' > i$, $\rho, i', \nu \models \psi'_2$ and for any point i'' between i and i', $\rho, i'', \nu \models \psi_1$.
   Let $i \leq j$ be any arbitrary point. Let $j_{min} > j$ be the point closest to j satisfying (I). Similarly, let i_m be any point satisfying (II). Either $i_m \leq j_{min}$ or $i_m > j_{min}$. In the former case, $\rho, i', \nu \models \psi_2$ iff $\rho, i', \nu \models \psi'_2$, by induction hypothesis. Hence, in the former case, (B2.1) iff (B2.2). Now, suppose $i_m > j_{min}$. Then notice that (as $i \leq j$, and j_{min} satisfies II), $j_{min} > i$, (IV-a) $\rho, j_{min}, \nu \models \psi_2$. Moreover, as i_m satisfies (II), all the points between i and i_m of ρ satisfies ψ_1 for the given valuation ν. As $j_{min} < i_m$, all the points

between i and j_{min} also lies between i and i_m. Hence, (V) for all points j'' between i and j_{min} $\rho, j'', \nu \models \psi_1$. By induction hypothesis, (IV-a) iff (IV-b), where (IV(b)) states $\rho, j_{min}, \nu \models \psi_2'$. Hence, (IV-a) and (V) iff (IV-b) and (V). Notice that (IV-b) and (V) is equivalent to j_{min} satisfying (III). Moreover, (IV-a) and (V) is equivalent to j_{min} satisfying (II). Hence, if $j_{min} < i_m$, then (B2.1) iff B2.2). Hence, (A) implies (B2.1) iff (B2.2), in both cases. Thus, (A) implies that for all $i \leq j$ (B) holds.

3. $\psi = \psi_1 \,\mathbf{S}\, \psi_2$. This case is symmetrical to the previous case (i.e., $\psi = \psi_1 \,\mathbf{U}\, \psi_2$). ◀

B Proof of Lemma 4.5

Proof. Let $\psi(x_1, \ldots, x_k)$ be a UPTL formula in CoT. We prove the lemma for $\leq \in \mathsf{Pol}(\varphi)$, and the argument is symmetric for $\geq \in \mathsf{Pol}(\varphi)$. The proof follows by structural induction on ψ.

Base Case. If ψ is a closed formula or a conjunction of atomic time constraints, then $\psi_1(x_{i_1}) \wedge \ldots \wedge \psi_k(x_{i_k})$ is the same as ψ. Hence, the lemma trivially holds.

Induction Hypothesis. Let the lemma hold for all the subformulas of ψ.

Induction. There are three possible cases.

- $\psi = \alpha \wedge \beta$- Notice that $\alpha{\downarrow_{c_j}} \wedge \beta{\downarrow_{c_j}} = \psi{\downarrow_{c_j}}(x_{i_j})$ for all $1 \leq j \leq k$. By induction hypothesis, $\alpha \equiv \alpha_1 \wedge \alpha_2 \ldots \alpha_k$, and $\beta \equiv \beta_1 \wedge \beta_2 \wedge \ldots \wedge \beta_k$. Hence, $\psi = \psi{\downarrow_{c_1}}(x_{i_1}) \wedge \ldots \wedge \psi{\downarrow_{c_k}}(x_{i_k})$.
- $\psi = \Box(\alpha)$ or $\psi = \overleftarrow{\Box}(\alpha)$. In this case, ψ is a closed formula as it is in CoT. Hence, the lemma trivially holds.
- $\psi = \alpha \,\mathbf{U}\, \beta$. Symmetric argument holds for $\psi = \alpha \,\mathbf{S}\, \beta$. Notice that as ψ is in CoT, α is a closed formula and β is in CoT. Hence, For any $1 \leq j \leq k$, $\psi{\downarrow_{c_j}} = \alpha \,\mathbf{U}\, \beta{\downarrow_{c_j}}$. By induction hypothesis $\beta = \beta{\downarrow_{c_1}} \wedge \ldots \wedge \beta{\downarrow_{c_k}}$. Hence, it suffices to show that for any timed word ρ, point g of ρ, and any valuation ν over variables of ψ, (A) $\rho, g, \nu \models \alpha \,\mathbf{U}\, (\beta{\downarrow_{c_1}} \wedge \ldots \wedge \beta{\downarrow_{c_k}})$ iff (B-1) $\wedge$ (B-2) $\wedge \ldots \wedge$ (B-k), where for all $1 \leq j \leq k$, (B-j) $\rho, g, \nu \models \alpha \,\mathbf{U}\, \beta{\downarrow_{c_j}}$. Notice that by semantics of $\mathbf{U}$, $\phi_1 \,\mathbf{U}\, (\phi_2 \wedge \phi_3)$ implies $\phi_1 \,\mathbf{U}\, (\phi_3)$ and $\phi_1 \,\mathbf{U}\, (\phi_2)$. Hence, (A) implies (B-1) $\wedge$ (B-2) $\wedge \ldots \wedge$ (B-k) holds trivially. The other direction is more interesting and relies on the untiming property of CoT. Hence, assume that (B-j) holds for all $1 \leq j \leq k$ (Assumption B). Thus for all $1 \leq j \leq k$, there exists a point g' such that (I-j) $g' > g$, $\rho, g', \nu \models \beta{\downarrow_{c_j}}$, and for all g'' between g and g', $\rho, g'', \nu \models \alpha$. For all $1 \leq j \leq k$, let g_j be any point in ρ satisfying (I-j). Let g_{min} be the point minimum among $g_1, \ldots, g_k$. Let $\beta_u = \mathsf{Untime}(\beta)$. Hence, β_u is a closed formula. Notice that for any $1 \leq j \leq k$, $\beta{\downarrow_{c_j}} \in \mathsf{Untime}(\beta)$ and $\beta_u \in \mathsf{Untime}(\beta{\downarrow_{c_j}})$. By Lemma 4.4 and Assumption B, $\rho, g_{min}, \nu \models \beta_u$. Again by By Lemma 4.4, Assumption B and $\rho, g_{min}, \nu \models \beta_u$, $\rho, g_{min}, \nu \models \beta{\downarrow_{c_j}}$ for all $1 \leq j \leq k$. Hence, there exists a point $g_{min} > g$ such that $\rho, g_{min}, \nu \models \beta{\downarrow_{c_1}} \wedge \beta{\downarrow_{c_2}} \ldots \wedge \beta{\downarrow_{c_k}}$, and for all points g'' between g and g_{min}, $\rho, g'', \nu \models \alpha$. Hence, (A) holds. ◀

C Proof of Lemma 4.6

This section is dedicated to the proof of the above lemma 4.6. We fix a UPTL formula φ and let $\bigvee \mathsf{CoT}^{\varphi}$denote the set of UPTL formulas such that $\mathsf{Pol}(\varphi) = \mathsf{Pol}(\psi)$ for all $\psi \in \bigvee \mathsf{CoT}^{\varphi}$. We now show how to construct a formula Θ in $\bigvee \mathsf{CoT}^{\varphi}$. Let ϕ be any UPTL formula. We apply induction on the structure of ϕ.

For the base case, assume ϕ is a UPTL formula which is either a closed formula or an atomic timing constraint of the form $x \sim u$. Then $\phi \in \mathcal{B}(F)$ where F is a finite set consisting of UPTL formulas whose subformulas are (i) either closed, or (ii) are open constraints of the form $x_i \sim u_i$ such that all of them have the same polarity. Hence, the lemma trivially holds.

Induction Hypothesis. Assume that the result holds for all subformulas of φ.

- **(Case 0).** $\varphi = \alpha \wedge \beta$ or $\varphi = \alpha \vee \beta$. Notice that α and β has the same polarity as φ. Hence, by induction hypothesis, α and β can be reduced to an equivalent $\bigvee_i \alpha_i$ and $\bigvee_j \beta_j$, respectively, where each α_i and β_j are in $\bigvee \mathsf{CoT}^{\varphi}$. Hence, if $\varphi = \alpha \vee \beta$, then the required formula in $\bigvee \mathsf{CoT}^{\varphi}$ is $\bigvee_i \alpha_i \vee \bigvee_j \beta_j$. Similarly, if $\varphi = \alpha \wedge \beta$, then $\varphi = \bigvee_i \bigvee_j (\alpha_i \wedge \beta_j)$ is the required formula in CoT, as each α_i and β_j are in CoT having the same polarity as φ.
- **(Case 1).** $\varphi = \alpha \,\mathbf{U}\, \beta$ or $\alpha \,\mathbf{S}\, \beta$.
- **(Case 2).** $\varphi = \Box\alpha$ or $\overleftarrow{\Box}\alpha$.

We observe a simplification which can be applied to cases 1, 2 using the induction hypothesis. By induction hypothesis, α and β can be reduced to $\alpha' = \bigvee_m \alpha_m$ and $\beta' = \bigvee_l \beta_l$ in $\bigvee \mathsf{CoT}^{\varphi}$, respectively. Thus, without loss of generality, we consider cases 1 and 2 to be of the forms $\varphi = \alpha' \,\mathbf{U}\, \beta'$ ($\alpha' \,\mathbf{S}\, \beta'$) and $\Box\alpha'$ ($\overleftarrow{\Box}\alpha'$) respectively.

For case 1, we just argue about $\mathbf{U}$ formulas. The argument for $\mathbf{S}$ is symmetrical. Notice that $\phi \,\mathbf{U}\, (\psi \vee \psi') \equiv (\phi \,\mathbf{U}\, \psi) \vee (\phi \,\mathbf{U}\, \psi)'$. Hence, φ in case 1 can be further simplified by pulling the top-level disjunctions of β'. Hence, $\varphi \equiv \bigvee_l (\alpha' \,\mathbf{U}\, \beta_l)$, where each β_l is in CoT having the same polarity as φ. As $\bigvee \mathsf{CoT}^{\varphi}$ is closed under disjunctions, it suffices to show that each $(\alpha' \,\mathbf{U}\, \beta_l)$ is in CoT. This is shown by proof of Lemma 4.7. Moreover, φ in case 2 can be reduced to φ in case 1 by the construction in the proof of Lemma 4.8.

C.1 Proof of Lemma 4.7: Case 1.2

This section is dedicated to the proof of 4.7.

Proof. We will argue about $\mathbf{U}$. The case for $\mathbf{S}$ is symmetrical. We apply induction on the number of α_is in φ that are open.

Base case. If there are no α_is open in φ then φ is, by definition, in $\bigvee \mathsf{CoT}^{\varphi}$.

Induction hypothesis. Suppose the lemma holds for the case where k of the α_i's are open formulas.

Induction. We show the lemma holds for formulas having $k+1$ open α_i formulas. Without loss of generality assume that α_1 is an open formula. Then, $\rho, i, \nu \models \varphi$ is equivalent to

(A) $\exists j > i$ such that $\rho, j, \nu \models \beta$, and $\forall i < j' < j$, $\rho, j', \nu \models \bigvee_{g=1}^{n} \alpha_g$.

- **Case 1.1** This case has been discussed in the main paper (see Section 4.2). Please see the Figure 4 for intuition.
- **Case 1.2** $\geq \in \mathsf{Pol}(\varphi)$.
  - Without loss of generality, we assume the $\alpha_1, \ldots, \alpha_{k+1}$ are open formulas, and rest all the formulas are closed.
  - (A) is equivalent to (C0) $\rho, i, \nu \models (\bigvee_{g=k+2}^{n})\alpha_g \,\mathbf{U}\, \beta$,
    or for some $1 \leq h \leq k+1$, (C1) there exists a point i' such that $\rho, i', \nu \models \alpha_h$, and (C2)for all $i < i'' < i'$, $\rho, i'', \nu \models \bigvee_{l=k+2}^{n} \alpha_l$, and (C3) $\rho, j, \nu \models \beta$.
  - By Untiming Lemma 4.4, (C1) implies that for all $j' \geq i'$, α_h is equivalent to α'_h, where $\alpha'_h = \mathsf{Untime}(\alpha_h)$.
  - Hence, $\varphi \equiv \varphi_1 \vee \varphi_2$ where, $\varphi_1 = [(\bigvee_{g=k+2}^{n})\alpha_g) \,\mathbf{U}\, \beta], \varphi_2 = \bigvee_{h=1}^{k+1} [(\bigvee_{g=k+2}^{n} \alpha_g) \,\mathbf{U}\, (\alpha_h \wedge \varphi'_h]$, $\varphi'_h = ((\bigvee_{1 \leq g \leq n \wedge g \neq h} \alpha_g) \vee \alpha'_h) \,\mathbf{U}\, \beta$, and $\alpha'_h = \mathsf{Untime}(\alpha_h)$. Hence, α'_h is a closed formula.

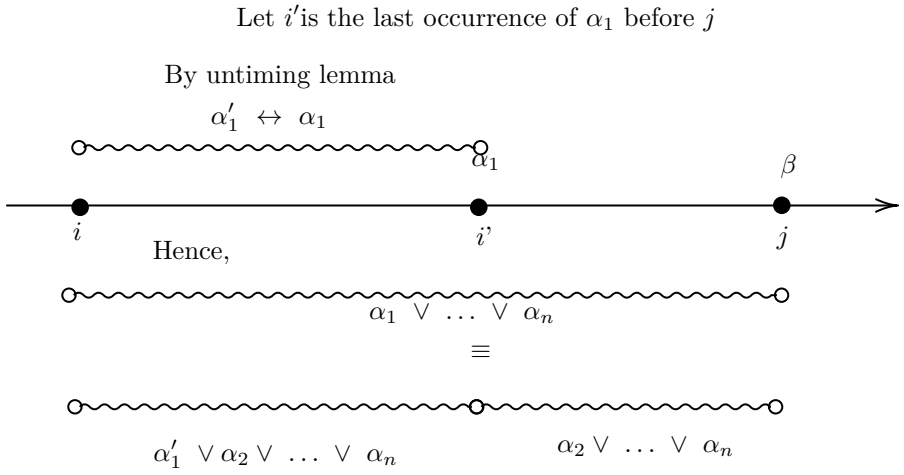


Figure 4 Figure Corresponding to the (B2) in Case 1.1 of Lemma 4.7.

- The left argument in $\mathbf{U}$ for φ_1 is closed (recall that $\alpha_1, \ldots, \alpha_{k+1}$ are the only open formulas) and the right argument (i.e. β) can be reduced to en equivalent formulas $\bigvee_g \beta_g$, where each β_g is in CoT. Hence, φ_1 is equivalent to Θ_1, where Θ_1 is $\Theta_1 = \bigvee_g [(\bigvee_{g=k+2}^{n})\alpha_g) \,\mathbf{U}\, \beta_g]$.
  Moreover φ'_h has one less open α_i subformula, as α'_h is closed. Hence, by induction hypothesis we can reduce φ'_h to an equivalent formula in $\bigvee \mathsf{CoT}^{\varphi}$, $\bigvee_l \psi_l$, where each ψ_l is in CoT and having the same polarity as φ. Pushing these disjunctions at the top, we get $\varphi_2 \equiv \Theta_2$ where $\Theta_2 = \bigvee_{h=1}^{k+1} \bigvee_l [(\bigvee_{g=k+2}^{n} \alpha_g) \,\mathbf{U}\, (\alpha'_h \wedge \psi_l]$.
  As $\bigvee \mathsf{CoT}^{\varphi}$ is closed under disjunction we just need to show that each $[(\bigvee_{g=k+2}^{n})\alpha_g) \,\mathbf{U}\, \beta_g]$ and $[(\bigvee_{g=k+2}^{n} \alpha_g) \,\mathbf{U}\, (\alpha'_h \wedge \psi_l]$ can be reduced to an equivalent formula in $\bigvee \mathsf{CoT}^{\varphi}$, which is implied by the induction hypothesis. ◀

D Proof of Lemma 4.8: Missing Cases

Notice that $\overleftarrow{\Box}(\bigvee_{g=1}^{n} \alpha_g) \equiv (\bigvee_{g=1}^{n} \alpha_g) \,\mathbf{S}\, (start)$, where $start = \overleftarrow{\Box} \bot$, is true only at the first point of the timed word. Hence, now applying Lemma 4.7, we get the required result.

Let us now consider $\varphi = \Box(\bigvee_{g=1}^{n} \alpha_i)$. We first discuss the Case 2.2, i.e. $\geq \in \mathsf{Pol}(\varphi)$, as it is not as involved as the Case 2.1 ($\leq \in \mathsf{Pol}(\varphi)$).

Case 2.2. Assume that $\geq \in \mathsf{Pol}(\varphi)$. We apply induction on the number of open α_is. If there are no open α_is we get a closed formula, and hence a $\bigvee \mathsf{CoT}^{\varphi}$ formula. Assume that for k open α_is, the lemma holds. We prove that the lemma holds for $k+1$ open α_is too.

- Without loss of generality we assume the $\alpha_1, \ldots, \alpha_{k+1}$ are open formulas, and the rest $\alpha_{k+2}, \ldots, \alpha_n$ are all closed.
- Then $\rho, i, \nu \models \varphi$ is equivalent to (C0) $\rho, i, \nu \models \Box[\bigvee_{g=k+2}^{n} \alpha_g]$, or,

- (C1) for some $1 \leq h \leq k+1$, there exists i' such that $\rho, i', \nu \models \alpha_h$, and (C2) for all $i < i'' < i'$, $\rho, i'', \nu \models \bigvee_{l=k+2}^{n} \alpha_l$, and (C3) $\rho, i', \nu \models \varphi = \Box(\bigvee_{g=1}^{n} \alpha_g)$.
- By untiming Lemma 4.4, (C1) implies that for all $j' \geq i'$, α_h is equivalent to α'_h, where $\alpha'_h = \mathsf{Untime}(\alpha_h)$.
- Hence, $\varphi \equiv \varphi_1 \vee \varphi_2$ where $\varphi_1 = [\Box(\bigvee_{g=k+2}^{n})\alpha_g)], \varphi_2 = \bigvee_{h=1}^{k+1}[(\bigvee_{g=k+2}^{n} \alpha_g)\,\mathbf{U}\,(\alpha_h \wedge \varphi'_h]$, $\varphi'_h = \Box(\bigvee_{1 \leq g \leq n \wedge g \neq h}(\alpha_g) \vee \alpha'_h)$, and $\alpha'_h = \mathsf{Untime}(\alpha_h)$. Notice that α'_h is a closed formula.
- Hence, φ'_h has at most k open α_is in its argument. Thus, by induction hypothesis, each φ'_h can be reduced into equivalent $\bigvee_l \psi_{h,l}$ where each $\psi_{h,l}$ is in CoT having same polarity as φ. Thus, φ_2 is equivalent to $\Theta_3 = \bigvee_{h=1}^{k+1} \bigvee_l [(\bigvee_{g=k+2}^{n} \alpha_g)\,\mathbf{U}\,(\alpha_h \wedge \psi_{h,l}]$. As $\bigvee \mathsf{CoT}^{\varphi}$ is closed under finite disjunctions, we just need to show that each $[(\bigvee_{g=k+2}^{n} \alpha_g)\,\mathbf{U}\,(\alpha_h \wedge \psi_{h,l}]$ can be reduced to an equivalent $\bigvee \mathsf{CoT}^{\varphi}$ formula. This is implied by Lemma 4.7.

Case 2.1. $\leq \in \mathsf{Pol}(\varphi)$. This case has already been discussed in the main paper. There are only couple of cases (Case A and Case B) left while arguing about the general case of infinite words in Section 4.3.1. We just discuss those cases here.

D.1 Handling the infinite word case: Missing Cases

- Case A - α_1 is not satisfied by any point $j > i$ of ρ for the given valuation ν. In this case, φ is equivalent to $\alpha_2 \vee \ldots \vee \alpha_n$ holds globally after i. Hence,
  - $\varphi_1 = \Box(\alpha_2 \vee \ldots \vee \alpha_n)$

  As φ_1 is smaller than φ, by induction hypothesis we can construct a formula $\delta = \bigvee_k \delta_k$, where each δ_k is in CoT.
- Case B - α_1 is finitely occurring in ρ, ν and is satisfied by some point $j > i$ for a given valuation ν. Then, φ implies α' holds until there exists a point in the future, where α_1 holds (possibly the last time) after which $\alpha_2 \vee \ldots \vee \alpha_n$ holds globally. Hence, we get $\varphi' = \alpha'\,\mathbf{U}\,\{\alpha_1 \wedge (\varphi_1)\}$. Substituting for φ_1 in φ' we get:
  - $\varphi_2 = \bigvee_k [\alpha'\,\mathbf{U}\,\{\alpha_1 \wedge \delta_k\}]$

  φ_2 falls in the category of case 1.1 of Lemma 4.7. Hence, we can construct an equivalent formula ζ in $\bigvee \mathsf{CoT}^{\varphi}$ equivalent to it.

E Past Modalities Add Expressiveness

▶ **Corollary E.1.** UMITPPL *is strictly more expressive than* UPTL[Fut].

Proof. By Lemma 3.4, UPTL is as expressive as 1-UPTL. It is known that 1-TPTL (and hence the subclass 1-UPTL) are less expressive than 1-clock Alternating Timed Automata (1-ATA) [19, 23, 24]. There exists a timed language L over a set of propositions $\{st, a, b\}$ that is not expressible in 1-ATA [25]. We write a UMITL formula expressing this language L, proving the result.

L consists of all timed words ρ for which (1) ρ begins with (st, τ_1), that is, proposition st is true at the first point of ρ with timestamp τ_1, (2) there are exactly two points in the next unit interval $(0, 1)$ from τ_1 with timestamps τ_2, τ_3 and a is true at both these points, and (3)

there is a point in ρ with timestamp $\tau_j \in [\tau_2 + 1, \tau_3 + 1]$ where b holds. This is captured in MITL by the formula:

$$\varphi = \neg \Diamond_{(0,1)}\{\overleftarrow{\Diamond}(\overleftarrow{\Diamond}(\overleftarrow{\Diamond}(\top)))\} \wedge \Diamond_{(0,1)}\{\overleftarrow{\Diamond}(\overleftarrow{\Diamond}(\top))\} \wedge \Diamond[b \wedge \overleftarrow{\Diamond}_{[0,1]}\{a \wedge (\overleftarrow{\mathsf{O}}\ (\overleftarrow{\mathsf{O}}\ (st)))\}] \wedge \Diamond[b \wedge \overleftarrow{\Diamond}_{[1,2]}\{a \wedge (\overleftarrow{\mathsf{O}}\ (st))\}]$$

It can be seen that $[\![\varphi]\!] = L$. ◀

On-The-Fly Symbolic Algorithm for Timed ATL with Abstractions

Nicolaj Ø. Jensen
Department of Computer Science, Aalborg University, Denmark

Kim G. Larsen
Department of Computer Science, Aalborg University, Denmark

Didier Lime
Nantes Université, École Centrale Nantes, CNRS, LS2N, UMR 6004, F-44000 Nantes, France

Jiří Srba
Department of Computer Science, Aalborg University, Denmark

Abstract

Verification of real-time systems with multiple components controlled by multiple parties is a challenging task due to its computational complexity. We present an on-the-fly algorithm for verifying timed alternating-time temporal logic (TATL), a branching-time logic with quantifiers over outcomes that results from coalitions of players in such systems. We combine existing work on games and timed CTL verification in the abstract dependency graph (ADG) framework, which allows for easy creation of on-the-fly algorithms that only explore the state space as needed. In addition, we generalize the conventional inclusion check to the ADG framework which enables dynamic reductions of the dependency graph. Using the insights from the generalization, we present a novel abstraction that eliminates the need for inclusion checking altogether in our domain. We implement our algorithms in Uppaal and our experiments show that while inclusion checking considerably enhances performance, our abstraction provides even more significant improvements, almost two orders of magnitude faster than the naive method. In addition, we outperform Uppaal Tiga, which can verify only a strict subset of TATL. After implementing our new abstraction in Uppaal Tiga, we also improve its performance by almost an order of magnitude.

2012 ACM Subject Classification Software and its engineering → Model checking; Theory of computation → Dynamic programming

Keywords and phrases Timed ATL, Symbolic Algorithms, Dependency Graphs, Timed Games

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2025.25

Related Version *Full Version*: https://arxiv.org/abs/2506.07802

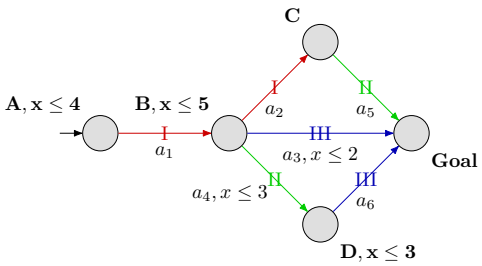
Supplementary Material
Software (Reprod. Package): https://doi.org/10.5281/zenodo.15195408

Funding VILLUM INVESTIGATOR project S4OS and ANR project BisoUS ANR-22-CE48-0012.

1 Introduction

Correctness is essential for many real-time systems, including distributed communication networks, energy grid management, and air traffic control. Ensuring the correctness of such systems is challenging due to the number of concurrent internal and external components. The interleaved behavior of all these components leads to an exponential growth in the number of states, a phenomenon known as the state-space explosion problem [14]. Addressing this challenge is a core focus of the field of model checking. In this paper, we consider real-time systems with components controlled by more than two parties, some cooperative, others adversarial. Real examples of such systems include electronic circuits with multiple components, perhaps some counterfeit [27], and networks of routers located in different

© Nicolaj Ø. Jensen, Kim G. Larsen, Didier Lime, and Jiří Srba;
licensed under Creative Commons License CC-BY 4.0



$\mathcal{A} \not\models \langle\!\langle \text{I} \rangle\!\rangle \Diamond \textbf{Goal}$
$\mathcal{A} \models \langle\!\langle \text{II} \rangle\!\rangle \Diamond \textbf{Goal}$
$\mathcal{A} \not\models \langle\!\langle \text{III} \rangle\!\rangle \Diamond \textbf{Goal}$
$\mathcal{A} \models \langle\!\langle \text{I}, \text{III} \rangle\!\rangle \Diamond \textbf{Goal}$
$\mathcal{A} \models [\![\text{II}]\!] \Diamond \textbf{Goal}$
$\mathcal{A} \not\models \langle\!\langle \text{II} \rangle\!\rangle (\neg \textbf{C} \,\mathcal{U}\, \textbf{Goal})$
$\mathcal{A} \models \langle\!\langle \text{I}, \text{III} \rangle\!\rangle (\neg \textbf{C} \,\mathcal{U}\, \textbf{Goal})$
$\mathcal{A} \models \langle\!\langle \text{I} \rangle\!\rangle \Box \neg \langle\!\langle \text{III} \rangle\!\rangle \Diamond \textbf{Goal}$
$\mathcal{A} \not\models \langle\!\langle \text{II} \rangle\!\rangle \Box \neg \langle\!\langle \text{III} \rangle\!\rangle \Diamond \textbf{Goal}$
$\mathcal{A} \not\models \langle\!\langle \text{II} \rangle\!\rangle \Diamond_{<5} \textbf{Goal}$

(a) An example timed game $\mathcal{A}$ with three players: I, II, and III and a single clock x that increases as time passes. Multiple TATL properties of the system are shown on the right.

$$\text{I} : \langle \ell, \nu \rangle \mapsto \begin{cases} a_1 & \text{if } \ell = \textbf{A} \\ a_2 & \text{if } \ell = \textbf{B} \wedge \nu(x) = 5 \\ \lambda & \text{otherwise} \end{cases} \qquad \text{III} : \langle \ell, \nu \rangle \mapsto \begin{cases} a_3 & \text{if } \ell = \textbf{B} \wedge \nu(x) \leq 2 \\ a_6 & \text{if } \ell = \textbf{D} \\ \lambda & \text{otherwise} \end{cases}$$

(b) An example strategy profile for player I and III that witnesses $\mathcal{A} \models \langle\!\langle \text{I}, \text{III} \rangle\!\rangle (\neg \textbf{C} \,\mathcal{U}\, \textbf{Goal})$.

Figure 1

countries [16, 6]. We model these systems as timed multiplayer games (TMG) and consider properties described in timed alternating-time temporal logic (TATL) [14, 4]. TMGs are generalized timed game automata [26, 7] and can have more than two players. Like in timed games, each discrete action belongs to a specific player and can be executed only at the discretion of that player. Consider the example TMG in Figure 1a with three players, I, II, and III, and a single real-valued clock x, initially at 0, that increases as time passes. From the starting location, **A**, player III has no strategy to ensure that the system reaches the **Goal** location, expressed as $\mathcal{A} \not\models \langle\!\langle \text{III} \rangle\!\rangle \Diamond \textbf{Goal}$ in TATL. This is because player I can wait in location **A** until $x > 3$ and then direct the system to location **C** without giving III any influence. Here, the system can remain indefinitely. On the other hand, II can ensure **Goal** is reached as I and III cannot leave the system in a location where II cannot make progress. TATL also allows us to consider coalitions of collaborating players. If players I and III work together, then they can ensure **Goal** is reached without visiting location **C**, expressed as $\mathcal{A} \models \langle\!\langle \text{I}, \text{III} \rangle\!\rangle (\neg \textbf{C} \,\mathcal{U}\, \textbf{Goal})$. We also consider nested properties, such as $\mathcal{A} \models \langle\!\langle \text{I} \rangle\!\rangle \Box \neg \langle\!\langle \text{III} \rangle\!\rangle \Diamond \textbf{Goal}$ which states that I can ensure that in all reached states, III does not have a strategy to ensure **Goal** is reached. Furthermore, timed properties add conditions on the time passed. For example, $\mathcal{A} \not\models \langle\!\langle \text{II} \rangle\!\rangle \Diamond_{<5} \textbf{Goal}$ states that II cannot guarantee **Goal** is reached strictly within 5 time units.

For verification, it is often unnecessary to explore the entire state space to determine the correctness of a property. One approach that reduces unnecessary work is to decompose the problem into sub-problems and lazily structure them in a dependency graph [25, 17, 18]. As the process unfolds, sub-problems eventually become trivial, and their solutions are used to resolve the more complex dependent problems. In many cases, the root problem can be answered without first constructing the entire dependency graph and the process can terminate early. This framework was formalized in [17, 18] using extended abstract dependency graphs (EADGs) and has been used to create on-the-fly algorithms for other model checking domains [12, 13, 20].

Our contributions

We develop an on-the-fly TATL algorithm within the EADG framework by providing a sound encoding of the problem. Our approach builds upon previous work involving encodings for timed systems and alternating-time temporal logic, showing their orthogonality, and we provide many semantic details that were left out in previous work. Since TATL is a superset of TCTL [1, 2, 4], our algorithm is also the first on-the-fly algorithm for TCTL with the generic freeze operator. Moreover, we formalize vertex merging in EADGs as a generalization of inclusion checking from prior work [13] and incorporate it into our encoding. In addition, this generalization also leads to an expansion abstraction that removes the need for conventional inclusion checking in our encoding altogether. Finally, we implement and evaluate various configurations of our algorithm and compare their performance to the state-of-the-art model checker for real-time games Uppaal Tiga [13, 9]. Our configuration with inclusion checking is, as expected, on par with Tiga on the subset of formulae that Tiga can handle, however, incorporating our expansion abstraction improves performance by nearly an order of magnitude.

Related work

The tool Uppaal Tiga [13, 9], which has since been integrated in Uppaal [19], uses a similar method to solve a simpler version of the problem: strategy synthesis in two-player timed games. In fact, [13] was published 20 years ago at CONCUR'05, and our paper brings the foundational work of that paper into the modern framework and expands it to the broader TATL logic. Tiga was the first to adapt the dependency graph-based on-the-fly algorithm by Liu and Smolka [25] to model checking, and the idea was eventually generalized into the EADG framework [17, 18]. Tiga relies on inclusion checking to merge vertices and improve performance, a detail that has not been generalized to the EADG framework until this paper. This paper also includes many semantic details that were left out in [13]. Alternating-time temporal logic (ATL), defined by T. Henzinger and R. Alur in [4] as an extension to branching-time logic [14], introduces ways to quantify over possible outcomes resulting from coalitions of players working together. ATL properties with a single coalition can be reduced to a synthesis problem in a two-player game, a problem that has already been intensively studied [26, 7, 15, 13]. However, it is also possible to nest the coalition quantifiers to express more intriguing properties. Recently, Carlsen et al. [12] used the EADG framework for on-the-fly verification of (untimed) ATL properties in concurrent games with multiple players. We extend this work to the setting of timed games and timed logics [1, 2]. An alternative approach to checking branching-time logic properties is the bottom-up algorithm [1, 14, 7]. This method begins by identifying all states that satisfy the relevant atomic propositions. It then iteratively propagates these results to compute states satisfying larger sub-properties until the original property is fully evaluated. While efficient when processing all reachable states is unavoidable, this approach may also waste time on unreachable states or irrelevant state-property combinations. Our on-the-fly algorithm mitigates these issues by computing only the necessary states and sub-properties on demand. Our new expansion abstraction relaxes the relevance criteria, bringing it closer to the bottom-up algorithm in this regard.

Paper structure

In Section 2, we present the TMG formalism and TATL logic followed by the general EADG framework. Then we present our encoding of the TATL problem in EADGs in Section 3. In Section 4, we introduce vertex merging to EADGs and an updated algorithm, allowing us

to perform inclusion checking in our encoding, and we introduce our expansion abstraction. Finally, in Section 5, we evaluate our algorithm and compare it against Uppaal Tiga on the problem instances where it is possible.

2 Preliminaries

We shall first introduce the model of timed games with multiple players as well as a timed logics for describing properties of such systems. Afterwards, we present the extended abstract dependency graph framework.

2.1 Timed Multiplayer Games

Let X be a finite set of real-valued variables called *clocks*. Let $C(X)$ be the set of *clock constraints* over the clocks X generated by the following abstract syntax:

$$g ::= x \bowtie k \mid x - y \bowtie k \mid g_1 \wedge g_2$$

where $k \in \mathbb{Z}$ and $x, y \in X$ and $\bowtie \in \{<, \leq, =, \geq, >\}$. Let $B(X) \subseteq C(X)$ be the subset that does not use any diagonal constraints of the form $x - y \bowtie k$. Let $\hat{B}(X) \subseteq B(X)$ be the subset that also does not use any constraints of the form $x < k$. A clock *valuation* $\nu : X \to \mathbb{R}_{\geq 0}$ assigns each clock a real value. The valuation denoted $\overrightarrow{0}$ assigns 0 to each clock. We write $\nu[Y]$ for a valuation that assigns 0 to any clock $x \in Y$ and assigns $\nu(x)$ to any clock $x \in X \setminus Y$. If $\delta \in \mathbb{R}_{\geq 0}$ then $\nu + \delta$ denotes the valuation such that $(\nu + \delta)(x) = \nu(v) + \delta$ for all $x \in X$. If $g \in C(X)$, we write $\nu \vDash g$ and say that ν satisfies g if replacing all clocks in g with their respective value in ν makes the expression evaluate to true.

▶ **Definition 1** (Timed Automaton). *A* timed automaton (TA) *is a 6-tuple* $\langle L, \ell_{\text{init}}, X, A, T, I\rangle$ *where* L *is a finite set of locations,* $\ell_{\text{init}} \in L$ *is the initial location,* A *is a set of actions,* X *is a finite set of real-valued clocks,* $T \subseteq L \times B(X) \times A \times 2^X \times L$ *is a finite set of edges each with a unique action from* A*, and* $I : L \to \hat{B}(X)$ *assigns an invariant to each location.*

The semantics of a TA $\mathcal{A} = \langle L, \ell_0, X, A, T, I\rangle$ can be described with a labeled transition system. The states are pairs $\langle \ell, \nu\rangle$ where $\ell \in L$ is a location and $\nu : X \to R_{\geq 0}$ is a valuation such that $\nu \vDash I(\ell)$ and Q is the set of all states. Transition labels are discrete actions or real-valued delays, i.e. $A \cup \mathbb{R}_{\geq 0}$, and the transition relation $\to$ is a union of the binary relations $\xrightarrow{a}$ and $\xrightarrow{\delta}$ defined over states as follows:

- $\langle \ell, \nu\rangle \xrightarrow{a} \langle \ell', \nu'\rangle$ if there exists an edge $\langle \ell, g, a, Y, \ell'\rangle \in T$ such that $v \vDash I(\ell) \wedge g$, $v' = v[Y]$, and $v' \vDash I(\ell')$, and
- $\langle \ell, \nu\rangle \xrightarrow{\delta} \langle \ell, \nu'\rangle$ if $\delta > 0$, $v \vDash I(\ell)$, $v' = v + \delta$, and $v' \vDash I(\ell)$.

A *run* in a TA is a sequence $\langle \ell_0, \nu_0\rangle \xrightarrow{t_0} \langle \ell_1, \nu_1\rangle \xrightarrow{t_1} \langle \ell_2, \nu_2\rangle \cdots$ such that $t_i \in A \cup \mathbb{R}_{\geq 0}$. A run is *maximal* if it contains an infinite number of discrete action transitions, ends with an infinite number of delay transitions that sum to ∞ (divergence), or no transition is possible in the final state (a deadlock). For the latter case, there is always a well-defined final state since invariants of the form $x < k$ are disallowed. The set $Runs_{\mathcal{A}}(\langle \ell, \nu\rangle)$ contains all maximal runs starting from $\langle \ell, \nu\rangle$. Given a run σ we write $\sigma[i]$ for the ith state $\langle \ell_i, \nu_i\rangle$.

▶ **Definition 2** (Timed Multiplayer Game). *A* timed multiplayer game (TMG) *with* N *players is a timed automaton where the actions* A *have been partitioned into* N *disjoint sets,* $A = A_1 \uplus \cdots \uplus A_N$*. We denote by* $\Sigma = \{1, \ldots, N\}$ *the set of players.*

Strategies and outcomes

A (memoryless) *strategy* for a player $p \in \Sigma$ is a function $f_p : Q \to A_p \cup \{\lambda\}$ that maps each state to λ or a discrete action that is enabled in the state and belongs to p. The strategy informs p what to do in the given state and λ is a special symbol indicating that player p should do nothing and wait. For states where a delay transition is not possible, a strategy can only output λ if the player has no enabled actions available. Let F_p be the set of all strategies for the player p. By ξ_S we denote a strategy profile that assigns to each player $p \in S \subseteq \Sigma$ a strategy $f_p \in F_p$, and $\Xi(S)$ is the set of all such strategy profiles over S. When players pick actions according to their strategies, it restricts the possible runs. We write $Out_{\mathcal{A}}(\xi_S, \langle \ell, \nu \rangle) \subseteq Runs_{\mathcal{A}}(\langle \ell, \nu \rangle)$ for the subset of *outcomes* (runs) induced when the players of S adhere to the strategies of ξ_S.

▶ **Definition 3** (Outcomes). *Let $\mathcal{A} = \langle L, \ell_{\text{init}}, X, A, T, I \rangle$ be a TMG with players $\Sigma = \{1, \ldots, N\}$, let $\langle \ell_0, \nu_0 \rangle$ be a state where $\nu_0 \vDash I(\ell_0)$, and let $\xi_S \in \Xi(S)$ be a strategy profile for $S \subseteq \Sigma$. A maximal run $\sigma = \langle \ell_0, \nu_0 \rangle \xrightarrow{t_0} \langle \ell_1, \nu_1 \rangle \xrightarrow{t_1} \langle \ell_2, \nu_2 \rangle \cdots \in Runs_{\mathcal{A}}(\langle \ell_0, \nu_0 \rangle)$ is in $Out_{\mathcal{A}}(\xi_S, \langle \ell_0, \nu_0 \rangle)$ iff the following conditions hold:*

- *for all $i \geq 0$ such that $t_i = \delta \in \mathbb{R}_{\geq 0}$ we have that $\xi_S(p)(\langle \ell_i, \nu_i + \delta' \rangle) = \lambda$ for all $0 \leq \delta' < \delta$ and all $p \in S$,*
- *for all $i \geq 0$ such that $t_i = a \in A$ we have either $a \notin \bigcup_{p \in S} A_p$ or $\xi_S(p)(\langle \ell_i, \nu_i \rangle) = a$ with $a \in A_p$ for some $p \in S$.*

Note that we allow Zeno behavior (an infinite subsequence consisting of no delays) from both the coalition S and the opposition $\overline{S}$.

2.2 Timed Alternating-Time Temporal Logic

The alternating-time temporal logic [4] is a logic that offers quantification over possible outcomes resulting from a coalition of players in a multiplayer game.

▶ **Definition 4** (Alternating-Time Temporal Logic). *Given a set of atomic propositions Π, a set of clocks X, and a set of players Σ, an* alternating-time temporal logic (ATL) *property conforms to the abstract syntax:*

$$\phi ::= \pi \mid g \mid \neg\phi_1 \mid \phi_1 \vee \phi_2 \mid \langle\!\langle S \rangle\!\rangle \bigcirc \phi_1 \mid \langle\!\langle S \rangle\!\rangle (\phi_1 \mathcal{U} \phi_2) \mid [\![S]\!] (\phi_1 \mathcal{U} \phi_2)$$

where $\pi \in \Pi$, $g \in C(X)$, and $S \subseteq \Sigma$ is a coalition of players.

Given a TMG $\mathcal{A}$ with players $\Sigma = \{1, \ldots, N\}$ and a function $Lab : L \to 2^{\Pi}$ labeling locations with atomic propositions, the satisfaction relation $\vDash$ over states and ATL properties is defined inductively as follows:

- $\langle \ell, \nu \rangle \vDash \pi$ iff $\pi \in Lab(\ell)$,
- $\langle \ell, \nu \rangle \vDash g$ iff $\nu \vDash g$,
- $\langle \ell, \nu \rangle \vDash \neg\phi$ iff $\langle \ell, \nu \rangle \nvDash \phi$,
- $\langle \ell, \nu \rangle \vDash \phi_1 \vee \phi_2$ iff $\langle \ell, \nu \rangle \vDash \phi_1$ or $\langle \ell, \nu \rangle \vDash \phi_2$,
- $\langle \ell, \nu \rangle \vDash \langle\!\langle S \rangle\!\rangle \bigcirc \phi_1$ iff there exists a strategy profile $\xi_S \in \Xi(S)$ such that for all maximal runs $\sigma = \langle \ell_0, \nu_0 \rangle \xrightarrow{t_0} \langle \ell_1, \nu_1 \rangle \xrightarrow{t_1} \langle \ell_2, \nu_2 \rangle \cdots \in Out_{\mathcal{A}}(\xi_S, \langle \ell, \nu \rangle)$ we have $\sigma[i+1] \vDash \phi_1$ where $i \geq 0$ is the smallest i such that $t_i \in A$ and such an i must exist,[1]

[1] The next operator is usually left out of timed systems, but it is well-defined with action-based semantics like here.

- $\langle \ell, \nu \rangle \vDash \langle\!\langle S \rangle\!\rangle (\phi_1 \mathcal{U} \phi_2)$ iff there exists a strategy profile $\xi_S \in \Xi(S)$ such that for all maximal runs $\sigma \in Out_{\mathcal{A}}(\xi_S, \langle \ell, \nu \rangle)$ we have $\sigma \vDash_r \phi_1 \mathcal{U} \phi_2$ (defined below),
- $\langle \ell, \nu \rangle \vDash [\![S]\!] (\phi_1 \mathcal{U} \phi_2)$ iff for all strategy profiles $\xi_S \in \Xi(S)$ there exists a maximal run $\sigma \in Out_{\mathcal{A}}(\xi_S, \langle \ell, \nu \rangle)$ such that $\sigma \vDash_r \phi_1 \mathcal{U} \phi_2$,

and for a maximal run $\sigma = \langle \ell_0, \nu_0 \rangle \xrightarrow{t_0} \langle \ell_1, \nu_1 \rangle \xrightarrow{t_1} \langle \ell_2, \nu_2 \rangle \cdots$ we have $\sigma \vDash_r \phi_1 \mathcal{U} \phi_2$ iff there exists an $i \geq 0$ such that:

- for all $j < i$:
  - if $t_j \in \mathbb{R}_{\geq 0}$ then for all $\delta \in [0, t_j)$ we have $\langle \ell_j, \nu_j + \delta \rangle \vDash \phi_1$, or
  - if $t_j \in A$ instead then $\langle \ell_j, \nu_j \rangle \vDash \phi_1$, and
- either $\langle \ell_i, \nu_i \rangle \vDash \phi_2$ or $t_i \in \mathbb{R}_{\geq 0}$ and there exists a $\delta \in [0, t_i)$ such that $\langle \ell_i, \nu_i + \delta \rangle \vDash \phi_2$, and for all $\delta' \in [0, \delta)$ we have $\langle \ell_i, \nu_i + \delta' \rangle \vDash \phi_1 \vee \phi_2$. We note that the disjunction $\phi_1 \vee \phi_2$ is necessary for timed big-step runs (instead of simply ϕ_1). See [5] for details.

We note that $[\![\emptyset]\!] \equiv \exists$ and $\langle\!\langle \emptyset \rangle\!\rangle \equiv \forall$ and thus ATL is a superset of CTL [4]. Other notable abbreviations and equivalences are $\pi \vee \neg\pi \equiv \mathbf{true}$ and $\langle\!\langle S \rangle\!\rangle (\mathbf{true} \mathcal{U} \phi) \equiv \langle\!\langle S \rangle\!\rangle \Diamond \phi$ and $\neg \langle\!\langle S \rangle\!\rangle \Diamond \phi \equiv [\![S]\!] \Box \neg \phi$ and $\neg \langle\!\langle S \rangle\!\rangle \bigcirc \phi \equiv [\![S]\!] \bigcirc \neg \phi$.

▶ **Definition 5** (Timed ATL). Timed alternating-time temporal logic (TATL) *is the timed extension of ATL and is defined as ATL but with a new freeze operator: $z.\phi$, where $z \in X$ is a* formula clock *and ϕ is another TATL property. Note that a formula clock z may not be used in the guards and invariants of the TMG. Given a state $\langle \ell, \nu \rangle \in Q$ we have $\langle \ell, \nu \rangle \vDash z.\phi$ iff $\langle \ell, \nu[z] \rangle \vDash \phi$.*

The TATL freeze operator is commonly used indirectly through the more intuitive temporal operator $\mathcal{U}_{\lhd k}$ (where $\lhd \in \{<, \leq\}$). For example, if z is a new clock that does not appear elsewhere in the model or the formula then:

$$\langle\!\langle S \rangle\!\rangle (\phi_1 \mathcal{U}_{\lhd k} \phi_2) \equiv z.\langle\!\langle S \rangle\!\rangle ((\phi_1 \wedge z \lhd k) \mathcal{U} \phi_2)$$

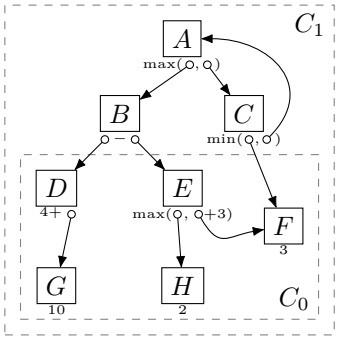
meaning that coalition $S \subseteq \Sigma$ has a strategy profile such that when adhered to, the system always reaches a state satisfying ϕ_2 (strictly) within k time units by a path where ϕ_1 invariantly holds.

An example strategy profile that witnesses $\mathcal{A} \vDash \langle\!\langle \mathrm{I}, \mathrm{III} \rangle\!\rangle (\neg \mathbf{C}\, \mathcal{U}\, \mathbf{Goal})$ for the TMG $\mathcal{A}$ in Figure 1a is given in Figure 1b. Since some states are never visited under this strategy, a partial strategy profile would also suffice as a witness. Such partial strategy witnesses enable our algorithm to terminate early for positive cases.

2.3 Extended Abstract Dependency Graphs

A *partially ordered set* $\langle \mathcal{D}, \sqsubseteq \rangle$ is a set $\mathcal{D}$ together with a binary relation $\sqsubseteq \subseteq \mathcal{D} \times \mathcal{D}$ that is reflexive, transitive and anti-symmetric. Given partially ordered sets $\langle \mathcal{D}_1, \sqsubseteq_1 \rangle$ and $\langle \mathcal{D}_2, \sqsubseteq_2 \rangle$, a function $f : \mathcal{D}_1 \to \mathcal{D}_2$ is *monotonically increasing* if $d \sqsubseteq_1 d'$ implies $f(d) \sqsubseteq_2 f(d')$ and *monotonically decreasing* if $d \sqsubseteq_1 d'$ implies $f(d) \sqsupseteq_2 f(d')$. When we just say *monotonic* we refer to increasing monotonicity. When a function has multiple inputs, monotonicity is with respect to a specific input. Unless we specify further, our monotonic multi-input functions are monotonic with respect to all inputs.

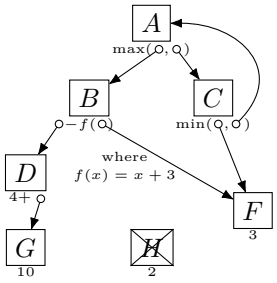
▶ **Definition 6** (Noetherian Ordering Relation). *A* Noetherian ordering relation with a least element (NOR) *is a triple $\langle \mathcal{D}, \sqsubseteq, \bot \rangle$ such that $\langle \mathcal{D}, \sqsubseteq \rangle$ is a partially ordered set, $\bot \in \mathcal{D}$ is the least element such that $\bot \sqsubseteq d$ for all $d \in \mathcal{D}$, and $\sqsubseteq$ satisfies the stabilizing ascending chain condition: for any chain $d_1 \sqsubseteq d_2 \sqsubseteq d_3 \sqsubseteq \cdots$ there exists a point i such that $d_i = d_j$ for all $j > i$.*



(a) Example EADG.

v	$\alpha^{C_0}_{\min}$	$\alpha^{C_1}_{\min}$
A	-	8
B	-	8
C	-	3
D	14	14
E	6	6
F	3	3
G	10	10
H	2	2

(b) Fixed points of components C_0 and C_1.



(c) The EADG after merge by $E \preceq_f F$ where $f(x) = x + 3$.

Figure 2 (a) An example EADG $\mathcal{G}$ over $\mathcal{D} = \langle \mathbb{N}_0 \cup \{\infty\}, \geq, \infty \rangle$. For each vertex v, the value function $\mathcal{E}(v)$ is displayed below the vertex, and the edges $E(v)$ are displayed using small circles inside the value functions, indicating the arity of the value function as well as which vertex assignment is used as input. Leaf nodes F, G, and H have no dependencies and their value function is a constant function. Since $\mathcal{E}(B)$ is non-monotonic, the graph has two components C_0 and C_1 which are highlighted with dashed boxes. (b) The fixed-point assignments of component C_0 and C_1 found by a fixed-point computation. (c) The EADG $\mathcal{G}[E \mapsto_f F]$ where vertex E has been removed through derivation $E \preceq_f F$. The value function of vertex B is pointwise composed with the derive function f on the second input which now points to the assignment of F instead. Vertex H now has no dependents and can be pruned.

▶ **Definition 7** (Abstract Dependency Graph). *An* abstract dependency graph (ADG) *is tuple* $\langle V, E, \mathcal{D}, \mathcal{E} \rangle$ *where* V *is a finite set of vertices,* $E : V \to V^*$ *is an edge function from vertices to strings of vertices,* $\mathcal{D}$ *is a NOR, and* $\mathcal{E}(v) : \mathcal{D}^n \to \mathcal{D}$ *is a monotonic value function at vertex* $v \in V$ *that takes* n *arguments where* $n = |E(v)|$.

We write $v \to v'$ if $v, v' \in V$ and $v' \in E(v)$, and $\to^+$ denotes the transitive closure of $\to$. The empty string of vertices is denoted ε. Vertices v where $E(v) = \varepsilon$ have constant value functions by definition.

▶ **Definition 8** (Extended Abstract Dependency Graph). *An* extended abstract dependency graph (EADG) *is a tuple* $\langle V, E, \mathcal{D}, \mathcal{E} \rangle$ *where* V, E, *and* $\mathcal{D}$ *are defined as for ADGs, but now* $\mathcal{E}(v) : \mathcal{D}^n \to \mathcal{D}$ *is a (possibly non-monotonic) value function at vertex* $v \in V$ *that takes* n *arguments with* $n = |E(v)|$. *Furthermore, if* $\mathcal{E}(v)$ *is non-monotonic, then* v *is not in a cycle, i.e. it is not the case that* $v \to^+ v$.

EADGs can be used to encode and solve various problems. Each vertex in the graph represents a problem and the NOR values $\mathcal{D}$ represent possible answers to problems with increasing accuracy. The value function at each vertex describes how the answer to the problem represented by the vertex can be derived from the answers to other (sub)problems. To describe this formally, we must partition the graph into components. Consider the function:

$$dist(v) = \max\{m \in \mathbb{N} \mid \exists v_0 v_1 \cdots v_n \in V^*, v = v_0, \forall i \in \{1, \ldots, n\}, v_{i-1} \to v_i, \\ m = |\{v_i \mid \mathcal{E}(v_i) \text{ is non-monotonic}\}|\} \tag{1}$$

describing how many non-monotonic value functions a vertex depends on. Since an EADG has no cycles with non-monotonic functions, $dist(v)$ is well-defined. The subgraphs induced by $dist$ are called components and $C_i \subseteq V$ is the ith component where $dist(v) \leq i$ for all $v \in C_i$. Component C_0 depends only on monotonic value functions.

Let $\alpha : V \to \mathcal{D}$ be an assignment from vertices to NOR values with $\alpha_\perp$ being the assignment such that $\alpha_\perp(v) = \perp$ for all vertices $v \in V$. Let F_0 be an update function such that:

$$F_0(\alpha)(v) = \mathcal{E}(v)(\alpha(v_1), \alpha(v_2), \ldots, \alpha(v_n)) \tag{2}$$

where $v \in C_0$ and $E(v) = v_1 v_2 \ldots v_n$. Since all value functions in C_0 are monotonic and $\mathcal{D}$ is a NOR, repeated application of F_0 on $\alpha_\perp$ eventually reaches a least fixed point, i.e. there exists an $m > 0$ such that $F_0^m(\alpha_\perp) = F_0^{m+1}(\alpha_\perp)$. We denote this fixed point $\alpha_{\min}^{C_0}$. For each component C_i where $i > 0$ the update function F_i is defined as follows:

$$F_i(\alpha)(v) = \begin{cases} \mathcal{E}(v)(\alpha(v_1), \alpha(v_2), \ldots, \alpha(v_n)) & \text{if } \mathcal{E}(v) \text{ is monotonic} \\ \mathcal{E}(v)(\alpha_{\min}^{C_{i-1}}(v_1), \alpha_{\min}^{C_{i-1}}(v_2), \ldots, \alpha_{\min}^{C_{i-1}}(v_n)) & \text{otherwise} \end{cases} \tag{3}$$

where $E(v) = v_1 v_2 \ldots v_n$ and the assignment $\alpha_{\min}^{C_{i-1}}$ is the minimal fixed point on component C_{i-1}, i.e. the fixed points are defined inductively [18]. Finally, let $C_{\max}$ be the component associated with the greatest *dist* in EADG G, and let $\alpha_{\min}^G$ denote $\alpha_{\min}^{C_{\max}}$. Given a vertex $v_0 \in V$, the value $\alpha_{\min}^G(v_0)$ can be efficiently found using the sound and complete on-the-fly algorithm introduced in [18].

A small example EADG can be seen in Figure 2a with the NOR $\mathcal{D} = \langle \mathbb{N}_0 \cup \{\infty\}, \geq, \infty \rangle$. In this domain, chains are descending, stabilizing before or at 0. In Figure 2b we show the minimum fixed-point assignment of the components in the example EADG.

3 Encoding TATL to EADGs

In this section we will introduce our symbolic encoding of the TATL problem to EADGs.

Symbolic Operations on Zones and Federations

Due to the uncountable size of the state space, automated verification of timed automata groups the concrete states into effectively representable sets of states and valuations [1, 10]. We shall now define some useful operators on such sets. Let $[\![g]\!] = \{\nu \in \mathbb{R}_{\geq 0}^X \mid \nu \vDash g\}$ be the set of all valuations satisfying $g \in C(X)$. A *zone* is a set $Z \subseteq \mathbb{R}_{\geq 0}^X$ of valuations where for some $g \in C(X)$ we have $[\![g]\!] = Z$. A finite union of zones is called a federation. If $W \subseteq \mathbb{R}_{\geq 0}^X$ is a set of valuations, then:

- $W^{\nearrow} = \{\nu + \delta \mid \nu \in W, \delta \in \mathbb{R}_{\geq 0}\}$ are the timed successors of W,
- $W^{\swarrow} = \{\nu' \in \mathbb{R}_{\geq 0}^X \mid \exists \nu \in W, \exists \delta \in \mathbb{R}_{\geq 0}, \nu' + \delta = \nu\}$ are the timed predecessors of W,
- $W[Y] = \{\nu[Y] \mid \nu \in W\}$ is a reset of the clocks $Y \subseteq X$ in W,
- $W\#x = \{\nu' \in \mathbb{R}_{\geq 0}^X \mid \exists \nu \in W, \forall y \in X, y \neq x \implies \nu(y) = \nu'(y)\}$ frees clock $x \in X$ in W.

Zones are closed under all the above operations [10]. Now consider a set of states $Q' \subseteq Q$ of a timed automaton $\langle L, \ell_{\text{init}}, X, A, T, I \rangle$. We extend these operations on clock valuations to the sets of states such that if $F \in \{\cdot^{\nearrow}, \cdot^{\swarrow}, \cdot[Y], \cdot\#x\}$ then $F(Q') = \{\langle \ell, \nu' \rangle \mid \exists \langle \ell, \nu \rangle \in Q', \nu' \in F(\{\nu\}) \cap [\![I(\ell)]\!]\}$. Given an action $a \in A$, we define the discrete a-predecessors and a-successors as $Pred_a(Q') = \{\langle \ell, \nu \rangle \mid \exists \langle \ell', \nu' \rangle \in Q', \langle \ell, \nu \rangle \xrightarrow{a} \langle \ell', \nu' \rangle\}$ and $Post_a(Q') = \{\langle \ell, \nu \rangle \mid \exists \langle \ell', \nu' \rangle \in Q', \langle \ell', \nu' \rangle \xrightarrow{a} \langle \ell, \nu \rangle\}$, respectively. Both $Pred_a$ and $Post_a$ preserve zones and federations [10]. When the automata is a TMG with players $\Sigma = \{1, \ldots, N\}$ and $S \subseteq \Sigma$ is a coalition, then we let A_S denote $\bigcup_{p \in S} A_p$. The S-controllable discrete predecessors of the coalition is defined by $Pred_S(Q') = \bigcup_{a \in A_S} Pred_a(Q')$. Similarly, the S-uncontrolled

discrete predecessors is $Pred_{\overline{S}}(Q')$ where $\overline{S} = \Sigma \setminus S$. Safe timed predecessors are those that avoid a set of states $Q'' \subseteq Q$ even as time elapses. As defined in [13]:

$$Pred_\lambda(Q', Q'') = \{\langle \ell, \nu - \delta \rangle \mid \langle \ell, \nu \rangle \in Q', \delta \geq 0, \forall \delta' \in [0, \delta], \langle \ell, \nu - \delta' \rangle \notin Q''\} \ .$$

The states of Q' where time cannot pass are defined as $Q'^{\nearrow} = \{\langle \ell, \nu \rangle \in Q' \mid \exists \text{"}x \leq k\text{"} \in I(\ell), \nu(x) = k\}$. These states are also called *time-locked*. We note that the $\cdot^{\nearrow}$ operator preserves federations. Finally, let $Zone(Q) \subseteq 2^Q$ be all subsets of Q that can be represented with a location-zone pair $\langle \ell, Z \rangle$ (also called a symbolic state), and let $Fed(Q) \subseteq 2^Q$ be all subsets of Q that can be represented with a location-federation pair $\langle \ell, \bigcup_i Z_i \rangle$.

The Encoding

We shall now encode the TATL problem on TMGs as an EADG G in order to answer the question: given a TMG $\mathcal{A} = \langle L, \ell_{\text{init}}, X, A, T, I \rangle$ with players $\Sigma = \{1, \ldots, N\}$ and a TATL formula ϕ_0, is it the case that $\langle \ell_{\text{init}}, \overrightarrow{0} \rangle \vDash \phi_0$? The vertices in EADG will have the form $\langle R, \phi \rangle$ where $R \in Zone(Q)$ is a symbolic state and ϕ is a TATL formula (a sub-formula of ϕ_0). We shall sometimes specify the location and zone of R and write $\langle \ell, Z, \phi \rangle$ for vertices instead, but the notation is often simpler for sets of states so we use R in those cases. Our root vertex is $\langle \ell_{\text{init}}, \{\overrightarrow{0}\}^{\nearrow} \cap [\![I(\ell_{\text{init}})]\!], \phi_0 \rangle$ and all dependencies of all vertices are generated using operators that preserve symbolic states. Without loss of generality, we assume all clocks are bounded[2] [13, 10] and hence there are only finitely many symbolic states in practice, and therefore the dependency graph is finite. Our assignment domain is the NOR $\langle Fed(Q), \subseteq, \emptyset \rangle$ and we typically denote elements from our NOR using W. We restrict ourselves such that $\alpha^G_{\text{min}}(\langle R, \phi \rangle) \subseteq R$ by construction and our goal is that $\langle \ell, \nu \rangle \in \alpha^G_{\text{min}}(\langle R, \phi \rangle)$ iff $\langle \ell, \nu \rangle \vDash \phi$. The value functions of our EADG rely on the helper function $Forceable_S(W_{\phi_1}, W_{\phi_2}, \mathcal{W})$ and its counterpart $Unavoidable_S(W_{\phi_1}, W_{\phi_2}, \mathcal{W})$ where S is a coalition of players, W_{ϕ_1} and W_{ϕ_2} are sets of states (typically associated with two TATL formulae), and $\mathcal{W}$ is a set of states. We have $\langle \ell, \nu \rangle \in Forceable_S(W_{\phi_1}, W_{\phi_2}, \mathcal{W})$ if and only if there exists a $\delta \geq 0$ such that $\langle \ell, \nu \rangle \xrightarrow{\delta} \langle \ell, \nu + \delta \rangle$ and

- for all $\delta' \in [0, \delta]$ either $\langle \ell, \nu + \delta' \rangle \in W_{\phi_2}$, or $\langle \ell, \nu + \delta' \rangle \in W_{\phi_1}$ and for all $a_i \in A_{\overline{S}}$ if $\langle \ell, \nu + \delta' \rangle \xrightarrow{a_i} \langle \ell', \nu' \rangle$ then $\langle \ell', \nu' \rangle \in \mathcal{W}$,
- and additionally
  - $\langle \ell, \nu + \delta \rangle \in W_{\phi_2}$, or
  - $\langle \ell, \nu + \delta \rangle \xrightarrow{a_j} \langle \ell', \nu' \rangle \in \mathcal{W}$ for some $a_j \in A_S$, or
  - $\langle \ell, \nu + \delta \rangle \not\xrightarrow{\delta''}$ for any $\delta'' > 0$ and $\langle \ell, \nu + \delta \rangle \xrightarrow{a_j} \langle \ell', \nu' \rangle \in \mathcal{W}$ for some $a_j \in A_{\overline{S}}$.

That is, the set is the subset of $W_{\phi_1} \cup W_{\phi_2}$ where coalition S can controllably get to W_{ϕ_2} with delays or to $\mathcal{W}$ with delays and a discrete action all while staying in W_{ϕ_1} until then. The set can be described symbolically as follows:

$$Forceable_S(W_{\phi_1}, W_{\phi_2}, \mathcal{W}) =$$
$$Pred_\lambda\left(W_{\phi_2} \cup Pred_S(\mathcal{W}) \cup \mathcal{H}, \left[\overline{W_{\phi_1}} \cup Pred_{\overline{S}}(\overline{\mathcal{W}})\right] \setminus W_{\phi_2}\right) \tag{4}$$
$$\text{where } \mathcal{H} = Q^{\nearrow} \cap Pred_{\overline{S}}(\mathcal{W}) \setminus Pred_\Sigma(\overline{\mathcal{W}}). \tag{5}$$

Similarly, we have $\langle \ell, \nu \rangle \in Unavoidable(W_{\phi_1}, W_{\phi_2}, \mathcal{W})$ if and only if there exists a $\delta \geq 0$ such that $\langle \ell, \nu \rangle \xrightarrow{\delta} \langle \ell, \nu + \delta \rangle$ and

[2] It is also possible to use extrapolation techniques instead [8].

- for all $\delta' \in [0, \delta)$ we have $\langle \ell, \nu + \delta' \rangle \in W_{\phi_1}$ and for all $a_i \in A_S$ if $\langle \ell, \nu + \delta' \rangle \xrightarrow{a_i} \langle \ell', \nu' \rangle$ then $\langle \ell', \nu' \rangle \in \mathcal{W}$,
- and additionally
  - $\langle \ell, \nu + \delta \rangle \in W_{\phi_2}$, or
  - $\langle \ell, \nu + \delta \rangle \in W_{\phi_1}$ and $\langle \ell, \nu + \delta \rangle \xrightarrow{a_j} \langle \ell', \nu' \rangle \in \mathcal{W}$ for some $a_j \in A_{\overline{S}}$, or
  - $\langle \ell, \nu + \delta \rangle \in W_{\phi_1}$ and $\langle \ell, \nu + \delta \rangle \xrightarrow{\delta''} \not\rightarrow$ for any $\delta'' > 0$ and for all $a_i \in A_S$ if $\langle \ell, \nu + \delta' \rangle \xrightarrow{a_i} \langle \ell', \nu' \rangle$ then $\langle \ell', \nu' \rangle \in \mathcal{W}$ and there exists at least one $a_i \in A_S$ such that $\langle \ell, \nu + \delta \rangle \xrightarrow{a_i}$.

That is, the set contains the subset of $W_{\phi_1} \cup W_{\phi_2}$ where coalition S cannot avoid that the system stays in W_{ϕ_1} and eventually reaches W_{ϕ_2} through delays, or reaches $\mathcal{W}$ through a delay followed by a discrete action. The set can be described symbolically as follows:

$$\begin{aligned} &\mathit{Unavoidable}_S(W_{\phi_1}, W_{\phi_2}, \mathcal{W}) = \\ &\quad \mathit{Pred}_\lambda\left(W_{\phi_2} \cup \mathit{Pred}_{\overline{S}}(\mathcal{W}) \cup \mathcal{H}, \left[\overline{W_{\phi_1}} \cup (\mathit{Pred}_S(\overline{\mathcal{W}}) \setminus \mathit{Pred}_{\overline{S}}(\mathcal{W}))\right] \setminus W_{\phi_2}\right) \qquad (6) \\ &\quad \text{where } \mathcal{H} = Q^{\nearrow} \cap \mathit{Pred}_S(\mathcal{W}). \qquad (7) \end{aligned}$$

Assuming a fixed ordering of actions $A = \{a_1, \ldots, a_n\}$, we now define $E(v)$ and $\mathcal{E}(v)$ of the EADG as follows based on the form of vertex v:

- Case $v = \langle R, \pi \rangle$: $E(v) = \varepsilon$ and $\mathcal{E}(v)(\varepsilon) = \{\langle \ell, \nu \rangle \in R \mid \pi \in \mathit{Lab}(\ell)\}$.
- Case $v = \langle R, g \rangle$: $E(v) = \varepsilon$ and $\mathcal{E}(v)(\varepsilon) = \{\langle \ell, \nu \rangle \in R \mid \nu \in [\![g]\!]\}$.
- Case $v = \langle R, \neg\phi \rangle$: $E(v) = \langle R, \phi \rangle$ and $\mathcal{E}(v)(W) = R \setminus W$.
- Case $v = \langle R, \phi_1 \vee \phi_2 \rangle$: $E(v) = \langle R, \phi_1 \rangle \langle R, \phi_2 \rangle$ and $\mathcal{E}(v)(W_1, W_2) = W_1 \cup W_2$.
- Case $v = \langle R, \phi_1 \wedge \phi_2 \rangle$: $E(v) = \langle R, \phi_1 \rangle \langle R, \phi_2 \rangle$ and $\mathcal{E}(v)(W_1, W_2) = W_1 \cap W_2$.
- Case $v = \langle R, z.\phi \rangle$: $E(v) = \langle R[z]^{\nearrow}, \phi \rangle$ and $\mathcal{E}(v)(W) = (W \cap Q[z])\#z \cap R$.
- Case $v = \langle R, \langle\!\langle S \rangle\!\rangle \bigcirc \phi \rangle$:

$$\begin{aligned} &E(v) = v_{a_1} \ldots v_{a_n} \quad \text{where } v_{a_i} = \langle \mathit{Post}_{a_i}(R)^{\nearrow}, \phi \rangle \\ &\mathcal{E}(v)(W_{a_1}, \ldots, W_{a_n}) = \mathit{Forceable}_S(R, \emptyset, \bigcup_i W_{a_i}) \end{aligned}$$

- Case $v = \langle R, \langle\!\langle S \rangle\!\rangle (\phi_1 \mathcal{U} \phi_2) \rangle$:

$$\begin{aligned} &E(v) = \langle R, \phi_1 \rangle \langle R, \phi_2 \rangle v_{a_1} \ldots v_{a_n} \quad \text{where } v_{a_i} = \langle \mathit{Post}_{a_i}(R)^{\nearrow}, \langle\!\langle S \rangle\!\rangle (\phi_1 \mathcal{U} \phi_2) \rangle \\ &\mathcal{E}(v)(W_{\phi_1}, W_{\phi_2}, W_{a_1}, \ldots, W_{a_n}) = \mathit{Forceable}_S(W_{\phi_1}, W_{\phi_2}, \bigcup_i W_{a_i}) \end{aligned}$$

- Case $v = \langle R, [\![S]\!] (\phi_1 \mathcal{U} \phi_2) \rangle$:

$$\begin{aligned} &E(v) = \langle R, \phi_1 \rangle \langle R, \phi_2 \rangle v_{a_1} \ldots v_{a_n} \quad \text{where } v_{a_i} = \langle \mathit{Post}_{a_i}(R)^{\nearrow}, [\![S]\!] (\phi_1 \mathcal{U} \phi_2) \rangle \\ &\mathcal{E}(v)(W_{\phi_1}, W_{\phi_2}, W_{a_1}, \ldots, W_{a_n}) = \mathit{Unavoidable}_S(W_{\phi_1}, W_{\phi_2}, \bigcup_i W_{a_i}) \end{aligned}$$

We remark that the elements of the NOR $\langle \mathit{Fed}(Q), \subseteq, \emptyset \rangle$ have finite representations using the data structure known as a *difference bound matrix* [10, 14]. All value functions are effectively computable since all constituent operations have known algorithms on difference bound matrices [10, 13].

In Figure 3, we show a fragment of the dependency graph generated when checking if $\phi = \langle\!\langle \mathrm{I}, \mathrm{III} \rangle\!\rangle (\neg \mathbf{C}\ \mathcal{U}\ \mathbf{Goal})$ holds in the initial state of $\mathcal{A}$ from Figure 1a. Eventually, the algorithm assigns $[\![x \leq 3]\!]$ to the vertex $\langle \mathbf{A}, x \leq 4, \phi \rangle$ and then we can terminate as we now know that the ϕ holds for the initial state $\langle \mathbf{A}, \overrightarrow{0} \rangle$.

We will now state the correctness of the encoding and its supporting lemma.

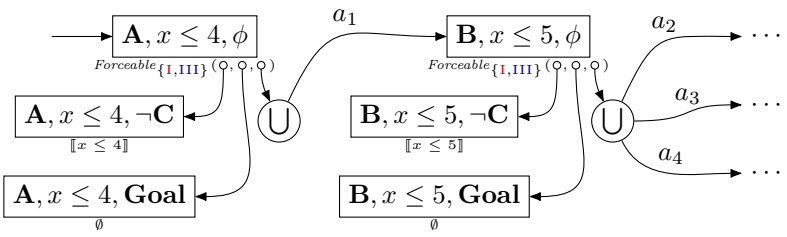


Figure 3 A fragment of the EADG for $\phi = \langle\!\langle \mathrm{I}, \mathrm{III} \rangle\!\rangle (\neg \mathbf{C}\, \mathcal{U}\, \mathbf{Goal})$ in $\mathcal{A}$ from Figure 1a.

▶ **Lemma 9** (Monotonically Safe). *The EADG encoding of TATL has no cycles of non-monotonic value functions and $dist(\langle R, \phi \rangle)$ is the greatest number of nested negations in ϕ.*

▶ **Theorem 10** (Encoding Correctness). *Given a TMG $\mathcal{A} = \langle L, \ell_{\text{init}}, X, A, T, I \rangle$, a set of states $R \in Zone(Q)$ such that $R = R^{\nearrow}$, a state $\langle \ell, \nu \rangle \in R$, and a TATL formula ϕ, then $\langle \ell, \nu \rangle \vDash \phi$ iff $\langle \ell, \nu \rangle \in \alpha^G_{\text{min}}(\langle R, \phi \rangle)$ using the presented EADG encoding of TATL.*

3.1 Unsatisfied states

We now detail an improvement also investigated in [13]. In the encoding presented above, the NOR domain represents the states for which the formula is definitely satisfied. It is also possible to define an encoding in which the NOR domain represents the states in which the formula is definitely not satisfied. It is easy to see that the composition of two NORs is also a NOR [17], so we can do both simultaneously. If the initial state is ever included in the set of unsatisfied valuations then we know that the property does not hold. Hence, with this extension, we can terminate early for many negative cases as well. The encoding using the unsat NOR is trivial for clock constraints $g \in C(X)$, and operators $\wedge$, $\vee$, and $\neg$, but for completeness, we state the encoding of formulae containing the $\bigcirc$ and $\mathcal{U}$ operator below. For all $v \in V$, we use the same edges $E(v)$, but different value functions, denoted $\mathcal{E}^c$, for updating the unsatisfied part of the composed NOR. Here M is used to denote elements in the unsat NOR and we assume the same ordering of actions $A = \{a_1, \ldots, a_n\}$ from earlier:

- Case $v = \langle R, \langle\!\langle S \rangle\!\rangle \bigcirc \phi \rangle$: $\mathcal{E}^c(v)(M_{a_1}, \ldots, M_{a_n}) = \mathit{Unavoidable}_S(R, \emptyset, \bigcup_i M_{a_i})$
- Case $v = \langle R, \langle\!\langle S \rangle\!\rangle (\phi_1 \mathcal{U} \phi_2) \rangle$:

$$\mathcal{E}^c(v)(M_{\phi_1}, M_{\phi_2}, M_{a_1}, \ldots, M_{a_n}) = \mathit{Unavoidable}_S(M_{\phi_2}, M_{\phi_1}, \bigcup_i M_{a_i})$$

- Case $v = \langle R, [\![S]\!] (\phi_1 \mathcal{U} \phi_2) \rangle$:

$$\mathcal{E}^c(v)(M_{\phi_1}, M_{\phi_2}, M_{a_1}, \ldots, M_{a_n}) = \mathit{Forceable}_S(M_{\phi_2}, M_{\phi_1}, \bigcup_i M_{a_i})$$

As seen, the difference between $\mathcal{E}$ and $\mathcal{E}^c$, is that every use of *Forceable* has been replaced by *Unavoidable*, and vice versa, and the terms involving ϕ_1 and ϕ_2 have been swapped in $\mathcal{U}$ formulae.

4 Dynamical Vertex Merge in EADGs

Usually, automated verification of timed systems takes advantage of inclusion checking, i.e. if one discovered symbolic state is included within another, then there is no reason to investigate the smaller one. To do this in the EADG framework, we must first introduce the general concept of vertex merging, which can be used to dynamically reduce the size of the graph.

Consider the NOR $\langle \mathcal{D}, \sqsubseteq, \bot \rangle$ and an EADG $G = \langle V, E, \mathcal{D}, \mathcal{E} \rangle$. If $u \in V^*$ is a string of vertices, then $u[i]$ denotes the ith vertex in the string and $u[v_1 \mapsto v_2]$ denotes the string u where all occurrences of v_1 are replaced with v_2. Given an n-ary function $h : \mathcal{D}^n \to \mathcal{D}$, a unary function $f : \mathcal{D} \to \mathcal{D}$ and an index i, $1 \leq i \leq n$, we use $\circ_i$ to denote pointwise function composition on the ith input, defined as $(h \circ_i f)(x_1, \ldots, x_n) = h(x_1, \ldots, f(x_i), \ldots, x_n)$, and its extension to multiple points $\circ_{\{i_1,\ldots,i_k\}}$ given by $(h \circ_{\{i_1,\ldots,i_k\}} f) = ((h \circ_{i_1} f) \circ_{i_2} \cdots) \circ_{i_k} f$ where notably $h \circ_\emptyset f = h$.

▶ **Definition 11** (Derivation). *Let $v_1, v_2 \in V$ be vertices in EADG $G = \langle V, E, \mathcal{D}, \mathcal{E} \rangle$ with $dist(v_1) \leq dist(v_2)$ and let $f : \mathcal{D} \to \mathcal{D}$ be a monotonic function such that $f(\alpha^G_{\min}(v_2)) = \alpha^G_{\min}(v_1)$. We call f a* derive function *and say that v_1 is* derivable *from v_2 through f, denoted by $v_1 \preceq_f v_2$.*

Derivable vertices can be removed by merging without loss of precision.

▶ **Definition 12** (Vertex Merge). *Let $v_1, v_2 \in V$ be vertices in EADG $G = \langle V, E, \mathcal{D}, \mathcal{E} \rangle$ with $dist(v_1) \leq dist(v_2)$ and let $f : \mathcal{D} \to \mathcal{D}$ be a monotonic function such that $v_1 \preceq_f v_2$. A* vertex merge of v_1 into v_2 by f *is an operation that results in a new EADG denoted $G[v_1 \mapsto_f v_2] = \langle V', E', \mathcal{D}, \mathcal{E}' \rangle$ where*

- *$V' = V \setminus \{v_1\}$,*
- *$E'(v) = E(v)[v_1 \mapsto v_2]$ for all $v \in V'$, and*
- *$\mathcal{E}'(v) = \mathcal{E}(v) \circ_I f$ for all $v \in V'$ where $I = \{i \mid E(v)[i] = v_1\}$.*

We can now present a theorem that shows that a vertex merge does not change the minimum fixed-point value of any vertex in the dependency graph.

▶ **Theorem 13** (Merge Preserves $\alpha^G_{\min}$). *Let $G = \langle V, E, \mathcal{D}, \mathcal{E} \rangle$ be an EADG. If $v_1, v_2 \in V$ and $f : \mathcal{D} \to \mathcal{D}$ such that $v_1 \preceq_f v_2$ then $\alpha^G_{\min}(v) = \alpha^{G[v_1 \mapsto_f v_2]}_{\min}(v)$ for all $v \in V \setminus \{v_1\}$.*

There are two main reasons why merging vertices can be beneficial when $v_1 \preceq_f v_2$. First, f may be computationally cheaper than computing $\mathcal{E}(v_1)$. Second, it reduces the size of the graph, making other operations such as back-propagation of updates and pruning cheaper. However, if f is expensive, it may be better to keep v_1 for the memoization it provides.

On Figure 2c, we show how the EADG from Figure 2a looks after a merge by derivation $E \preceq_f F$ where $f(x) = x + 3$. The existence of this derivation depends on the user's domain knowledge. In this case, the user might know that the first input to the max function of E can never exceed 3, and hence its value can be derived from the second input F. As shown in the figure, the resulting graph is smaller and it is also possible to prune the vertex H.

4.1 Algorithm with Vertex Merging

Here we present our modification to the fixed-point algorithm from [18], enabling it to take advantage of vertex merging whenever $v_1 \preceq_f v_2$. The updated algorithm uses the following data structures:

- $\hat{v}$ is the current root vertex, initialized to v_0,
- $\hat{f}$ is the root deriving function, initialized to id (the identity function),
- $\alpha : V \to \mathcal{D}$ is the current assignment, initialized to $\alpha_\bot$,
- W is a waiting set of vertices pending exploration or reevaluation,
- $Pass$ is a set of explored vertices,
- $Dep : V \to 2^V$ is a function that for each vertex v returns a set of dependent vertices that should be reevaluated if the assignment of v changes,

Algorithm 1 Minimum fixed-point computation on an EADG. The lines highlighted in gray are our additions to the algorithm from [18].

Input: An EADG $\langle V, E, \mathcal{D}, \mathcal{E} \rangle$ and $v_0 \in V$
Output: $d \in \mathcal{D}$ s.t. $d \sqsupseteq \alpha_{\min}(v_0)$

```
1  Dep(v) = ∅, Edges(v) := E(v), and Eval(v) = E(v) for all v ∈ V;
2  v̂ := v0; f̂ = id; α := α⊥;
3  W := {v0}; Pass := ∅; Active := {v0};
4  while W ≠ ∅ do
5      let v ∈ W where v is pickable;
6      W := W \ {v};
7      if v = v̂ or Dep(v) ≠ ∅ then
8          if v ∉ Pass then
9              v := Explore(v);
10         if v ∈ Pass or E(v) is non-monotonic then
11             let v1 v2 ... vk := Edges(v);
12             d := E(v)(α(v1), α(v2), ..., α(vk));
13             if α(v) ⊏ d then
14                 W := W ∪ {u ∈ Dep(v) | ∃i.Edges(u)[i] = v ∧ i ∉ Ignore(α, u)};
15                 α(v) := d;
16                 if v = v̂ and {1, 2, ..., k} ⊆ Ignore(α, v̂) then
17                     return f̂(α(v̂))
18         Pass := Pass ∪ {v};
19 return f̂(α(v̂))
20 proc Explore(v) :
21     v* = v;
22     for i = 1 to |Edges(v)| do
23         let vi = Edges(v)[i] (generate it if needed);
24         if vi ∉ Active then
25             Active := Active ∪ {vi};
26             forall v' ∈ Active where v' ≠ vi do
27                 if vi ⪯f v' for some f then
28                     Active := Active \ {vi};
29                     Eval(v) := Eval(v) ∘i f;
30                     Edges(v) := Edges(v)[vi ↦ v'];
31                     continue outer loop;
32                 if v' ⪯f vi for some f then
33                     Replace(v', f, vi);
34                     if v' = v̂ then
35                         v̂ := vi;
36                         f̂ := f̂ ∘ f;
37                     if v' = v* then
                           /* We replaced the vertex we were exploring.  Record replacement.  */
38                         v* := vi;
39         if v* ≠ v then
40             return Explore(vi)
41         Dep(vi) := Dep(vi) ∪ {v};
42         if vi ∉ Pass then W := W ∪ {vi};
43     return v
44 proc Replace(v1, f, v2) :
45     Dep(v2) := Dep(v1);
46     forall v' ∈ Active ∩ Dep(v1) do
47         Eval(v') := Eval(v') ∘I f where I = {i | Edges(v')[i] = v1};
48         Edges(v') := Edges(v')[v1 ↦ v2]
49     Pass := Pass \ {v1};
50     Active := Active \ {v1};
51     W := W \ {v1};
```

- $Edges : V \to V^*$ is a function that for each vertex v returns the current edges of v, initialized to $E(v)$,
- $Eval(v) : \mathcal{D}^n \to \mathcal{D}$ is the current value function for vertex v where $n = |E(v)|$, initialized to $\mathcal{E}(v)$,
- *Active* is a set of vertices that are (still) relevant for the root node $\hat{v}$ and considered during vertex merging.

The variables $\hat{v}$ and $\hat{f}$, and the data structures *Edges*, *Eval*(v), and *Active* are introduced by us to keep track of the reduced graph. In Algorithm 1, we show the EADG minimum fixed point computation algorithm from [18] updated with vertex merging.[3] Incorporating vertex merging is relatively straightforward, but there are some edge cases to discuss. Vertex merging, highlighted in gray in Algorithm 1, occurs during the exploration of the vertex v (line 20-43). Each dependency v_i of v is checked against all other active vertices. If v_i can be merged into another active vertex v', then we make v depend on v' instead and make v_i inactive (lines 27-31). If an active vertex v' can be merged into v_i then we make v' inactive and replace v' with v_i in all our data structures (line 33). Additionally, if v' is the root vertex $\hat{v}$ then we make v_i the new root and update $\hat{f}$ by composing it with f so that the root assignment can be derived later (lines 34-36). In (presumably) rare cases, v' is the explored vertex v, and if so, exploration must be restarted (lines 37-40).

▶ **Lemma 14.** *At the start and end of the* Explore *procedure, there exists no* $v_1, v_2 \in$ *Active such that* $v_1 \neq v_2$ *and* $v_1 \preceq_f v_2$.

▶ **Theorem 15.** *The algorithm terminates and returns* $\alpha^G_{\min}(v_0)$.

4.2 Inclusion Checking as Derivation

Finally, we can do vertex merging in our problem domain. If one symbolic state is included within another and the associated query is the same, we merge the smaller one into the bigger one, since it contains the same valuations and more (conventional inclusion checking).

▶ **Theorem 16** (Derivation by Intersection)**.** *Let* $\langle R, \phi\rangle, \langle R', \phi\rangle \in V$ *be vertices and let* $f_R : \mathcal{D} \to \mathcal{D}$ *be a function such that* $f_R(W) = W \cap R$. *If* $R \subseteq R'$, *then* $\langle R, \phi\rangle \preceq_{f_R} \langle R', \phi\rangle$ *is a derivation.*

4.3 Expansion Abstraction

It is known that EADGs can be refined using abstractions [18, 21, 22]. We now present a simple abstraction for our domain that has superior performance compared to the inclusion checking. As discussed, R is a symbolic state and $\langle \ell, Z, \phi\rangle$ is the actual form of vertices in the dependency graph. For any vertex $v = \langle \ell, Z, \phi\rangle$ there exists a vertex $v' = \langle \ell, [\![I(\ell)]\!], \phi\rangle$ and v can be merged into v' if it is generated. v' will not be merged into another vertex as it is the biggest w.r.t. our derivation function in Theorem 16. Assuming that v' has a high chance of being generated whenever v is, it makes sense to use an abstraction that takes v directly to v' immediately, as the merge is likely to happen later anyway, and their value functions require a similar number of operations but v' holds more information. Doing so for all vertices induces a new smaller EADG where inclusion checking and merging are unnecessary, as for every location there is a unique zone to be explored. In other words, our abstracted vertices are defined only by the discrete part of the state. If the assumption is wrong, i.e. v' is not discovered although v was, we may end up exploring states and edges that are not relevant in practice as more discrete transitions may be enabled in the the zone $[\![I(\ell)]\!]$ than in Z. However, we assess that timed automata with useless edges are rare in real use cases, and most valuations are eventually reached. As a consequence, the downside of this abstraction is insignificant, as also demonstrated by our experiments.

[3] We also rearranged and renamed some procedures for clarity and omitted the pruning procedure. Additionally, Ignore returns a set of indices instead of vertices in our version.

Formally, we apply the following *expansion abstraction* $\mathfrak{X} : V \to V$ to each vertex upon generation:

$$\mathfrak{X}(\langle \ell, Z, \phi \rangle) = \langle \ell, [\![I(\ell)]\!], \phi \rangle \ . \tag{8}$$

▶ **Theorem 17.** *Applying abstraction $\mathfrak{X}$ from Equation (8) to all vertices preserves encoding correctness, i.e. $\langle \ell, \nu \rangle \vDash \phi$ iff $\langle \ell, \nu \rangle \in \alpha_{\min}^{\mathfrak{X}(G)}(\langle R, \phi \rangle)$ for every $\langle \ell, \nu \rangle \in R$.*

If follows, that the vertex merging using the derivation we defined in Theorem 16 becomes redundant when using $\mathfrak{X}$ because there is only one vertex per location-formula pair. The forward exploration done by the algorithm now only considers whether there exists a valuation such that a location can transition another location. This is less strict than a traditional reachability analysis, but it avoids expensive inclusion checks of zones while still restricting the backward propagation to the location-formula pairs we care about.

5 Implementation and Experimental Evaluation

We implement our algorithm in Uppaal [19]. We use two stacks for the waiting list, one for vertices waiting for exploration and one for vertices waiting for updates. We always prioritize vertices from the update stack over vertices from the exploration stack, as this was found to be favorable in [24]. To improve performance, the EADG framework also allows us to specify a function Ignore(α, v) describing which vertices of $E(v)$ can be skipped given the current assignment α. We use the universally sound ignore function suggested in [18] by leveraging that $\alpha(\langle R, \phi \rangle) \subseteq R$. We can reduce the number of non-monotonic functions and thus graph components by rewriting the TATL formulae and pushing negations downward to the atomic propositions when possible.

The Benchmark

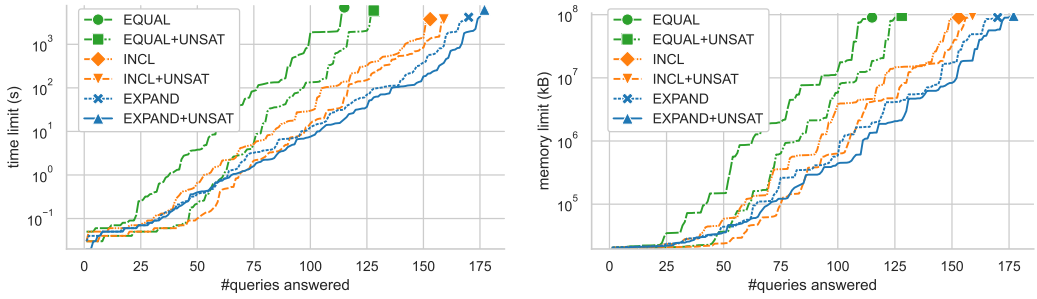
We benchmark various configurations:

- Equal: The traditional EADG algorithm without vertex merging.
- Incl: We use vertex merging through inclusion checking as by Theorem 16.
- Expand: We use the expansion abstraction $\mathfrak{X}$ described in Section 4.3 (and no vertex merging).
- +Unsat: We additionally compute unsatisfied states as described in Section 3.1.
- Tiga(+Unsat): The `verifyta` engine distributed with Uppaal version 5.0.0 [9, 19] and specifically the Tiga algorithm.[4]

The benchmark suite consists of 3 types of TMG models, each instantiated in various sizes and with multiple associated TATL properties, totaling 236 model and query executions. The models are:

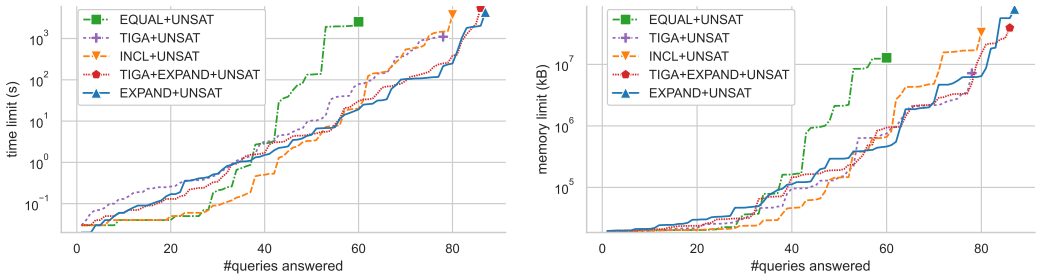
- Train Gate [3]: N trains share the same bridge and a controller signals the trains if the bridge is occupied. Each train and the controller is a different player. Example queries include:
  - $\langle\!\langle T_1, T_2, T_3 \rangle\!\rangle \Diamond num_crossing \geq 1$; Can train T_1, T_2, and T_3 create a situation where more than one train is crossing at the same time? (No)
  - $\langle\!\langle T_1 \rangle\!\rangle \Diamond crossed(T_1)$; Can train T_1 ensure that it eventually crosses the bridge? (No)

[4] Tiga always keeps track of losing states without a strategy, i.e. the Unsat modification.



(a) Time usage. **(b)** Memory usage.

Figure 4 Cactus plots showing how many queries are answered within the indicated per-query resource limit. A configuration performs better if it can answer more queries using fewer resources.



(a) Time usage. **(b)** Memory usage.

Figure 5 Cactus plots showing how many TIGA-compatible queries are answered within the indicated per-query resource limit. A configuration performs better if it can answer more queries using fewer resources.

- Mexican Standoff [12] extended with reload time: N cowboys are shooting at each other and must reload their gun between each shot. Each cowboy is a different player and the gun reloading system is also a player. Example queries include:
  - $\langle\!\langle C_1 \rangle\!\rangle (alive(C_1)\,\mathcal{U}\, t > 1)$; Can cowboy C_1 guarantee staying alive for 1 second? (No)
  - $\langle\!\langle Guns, C_1, \ldots, C_{\lceil N/2 \rceil} \rangle\!\rangle \Box \neg \bigvee_{c \in \{C_1, \ldots, C_{\lceil N/2 \rceil}\}} alive(c)$; If half the cowboys cooperate and the gun reloading is on their side, can they ensure one of them survives? (Yes)
- Phase King [11]: A consensus algorithm for N nodes where time is divided into phases and rounds, and the king of each phase decides the voting tiebreaker. Each node is controlled by a unique player. Example queries include:
  - $[\![\,]\!] \Diamond consensus(n_1, n_2, n_3)$; Node n_1, n_2, n_3 will reach consensus in some trace? (Yes)
  - $\langle\!\langle \rangle\!\rangle \Diamond consensus(n_1, n_2, n_3)$; Node n_1, n_2, n_3 will reach consensus in all traces? (No)
  - $\langle\!\langle \rangle\!\rangle \Box(\neg consensus(n_1, n_2, n_3) \vee \langle\!\langle n_1, n_2, n_3 \rangle\!\rangle \Box consensus(n_1, n_2, n_3))$; If nodes n_1, n_2, and n_3 have consensus, can they can stay in consensus? (Yes, if $N \leq 5$)

Evaluation

Performance comparison of the configurations can be seen in Figure 4. Here the query executions are ordered (independently for each method) by their running time. Note that the y-axis is logarithmic. The INCL configuration using vertex merging is more than one order of magnitude faster than the EQUAL configuration using no merging, and EXPAND is almost

another order of magnitude faster for the most difficult instances. The UNSAT modification further improves the three configurations matching the conclusion in [13]. We find a similar improvements for memory usage. As described in Section 4.3, EXPAND may explore edges unnecessarily, since the abstraction enables all transitions in a location, assuming they will be relevant eventually. In the plots, we see that the EXPAND configuration performs worse on easy queries since these extra edges hinder early termination. However, after just 1 second this downside is eliminated by the advantages of the method.

Of our 236 total queries, 150 have no nested coalitions and can be solved by UPPAAL TIGA as well. However, for a large portion of them, modifications to the TMG model are required to get the edges' controllability to correspond to the actions of the coalition. In Figure 5, we compare the performance with TIGA on this subset of 150 queries. As expected, INCL+UNSAT and TIGA+UNSAT have similar performance on challenging queries, as these two configurations are very similar in practice. Our implementation seems slightly faster on easy queries. When we add the EXPAND abstraction to TIGA and remove its now redundant inclusion checking (configuration TIGA+EXPAND+UNSAT), the runtime performance matches that of our EXPAND+UNSAT, almost an order of magnitude faster than the previous state-of-the-art TIGA implementation.

A reproducibility package is available at [23].

6 Conclusion

We presented an encoding of the timed alternating-timed temporal logic problem in the extended abstract dependency graph (EADG) framework. This involved combining previous work on timed games, timed CTL, and alternating-time temporal logic. Our work is thus an example of how various encodings of model-checking problems in the EADG framework are orthogonal and can be combined to solve the combined logic extensions. We also took this opportunity to provide many details left out in the previous paper on UPPAAL TIGA [13]. Furthermore, we formalized a generalization of conventional inclusion checking for the EADG framework. The resulting vertex merging technique can be used to remove vertices that can be derived from other vertices, which is especially useful when the derivation is cheaper than the computation of the value function of the removed vertex. Other domains where we foresee vertex merging being useful include model checking for Petri nets where one marking can cover other markings with less behavior. The vertex merging also allowed us to easily show that we can better exploit the symbolic representation of states using an expansion abstraction. In essence, our abstraction simplifies our symbolic states to discrete locations and their invariants. Hence, *all* valuations satisfying the given property in the location are propagated backward through the dependency graph, instead of restricting it to valuations that the exploration currently considers reachable. Thus, this abstraction makes the algorithm slightly closer to a traditional backward algorithm.

Our implementation and experiments showed that inclusion checking improves the performance of the naive encoding, while the expansion abstraction outperforms it by almost an additional order of magnitude. We also found that our algorithm is comparable to state-of-the-art UPPAAL TIGA when both use inclusion checking. By integrating our expansion abstraction in TIGA we also improved its performance by almost an order of magnitude. The algorithms presented in this paper will be made available in an upcoming release of UPPAAL.

References

1 R. Alur, C. Courcoubetis, and D. Dill. Model-checking for real-time systems. In *Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 414–425, 1990. doi:10.1109/LICS.1990.113766.

2 R. Alur, C. Courcoubetis, and D. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, 1993. doi:10.1006/inco.1993.1024.

3 Rajeev Alur. Timed automata. In Nicolas Halbwachs and Doron Peled, editors, *Computer Aided Verification*, volume 1633 of *LNCS*, pages 8–22, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. doi:10.1007/3-540-48683-6_3.

4 Rajeev Alur, Thomas A. Henzinger, and Orna Kupferman. Alternating-time temporal logic. *J. ACM*, 49(5):672–713, September 2002. doi:10.1145/585265.585270.

5 Hugh Anderson and P.S. Thiagarajan. Verification of real-time systems. Course material for CS5270 at NUS Computing, chapter 6, 2007.

6 Ruwaifa Anwar, Haseeb Niaz, David R. Choffnes, Ítalo F. S. Cunha, Phillipa Gill, and Ethan Katz-Bassett. Investigating interdomain routing policies in the wild. *Proceedings of the 2015 Internet Measurement Conference*, 2015. doi:10.1145/2815675.2815712.

7 Eugene Asarin, Oded Maler, Amir Pnueli, and Joseph Sifakis. Controller synthesis for timed automata. *IFAC Proceedings Volumes*, 31(18):447–452, 1998. 5th IFAC Conference on System Structure and Control 1998 (SSC'98), Nantes, France, 8-10 July. doi:10.1016/S1474-6670(17)42032-5.

8 Gerd Behrmann, Patricia Bouyer, Emmanuel Fleury, and Kim G. Larsen. Static guard analysis in timed automata verification. In Hubert Garavel and John Hatcliff, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2619 of *LNCS*, pages 254–270, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. doi:10.1007/3-540-36577-X_18.

9 Gerd Behrmann, Agnès Cougnard, Alexandre David, Emmanuel Fleury, Kim G. Larsen, and Didier Lime. Uppaal-tiga: Time for playing games! In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification*, LNCS, pages 121–125, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. doi:10.1007/978-3-540-7.

10 Johan Bengtsson and Wang Yi. *Timed Automata: Semantics, Algorithms and Tools*, pages 87–124. LNCS. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. doi:10.1007/978-3-540-27755-2_3.

11 Piotr Berman and Juan A. Garay. Cloture votes:n/4-resilient distributed consensus int + 1 rounds. *Mathematical systems theory*, 26(1):3–19, January 1993. doi:10.1007/BF01187072.

12 Falke B. Ø. Carlsen, Lars Bo P. Frydenskov, Nicolaj Ø. Jensen, Jener Rasmussen, Mathias M. Sørensen, Asger G. Weirsøe, Mathias C. Jensen, and Kim G. Larsen. Cgaal: Distributed on-the-fly ATL model checker with heuristics. *Electronic Proceedings in Theoretical Computer Science*, 390:99–114, September 2023. doi:10.4204/eptcs.390.7.

13 Franck Cassez, Alexandre David, Emmanuel Fleury, Kim G. Larsen, and Didier Lime. Efficient on-the-fly algorithms for the analysis of timed games. In Martín Abadi and Luca de Alfaro, editors, *CONCUR 2005 – Concurrency Theory*, volume 3653 of *LNCS*, pages 66–80, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. doi:10.1007/11539452_9.

14 Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem. *Handbook of Model Checking*. Springer Publishing Company, Incorporated, 1st edition, 2018. doi:10.1007/978-3-319-10575-8.

15 Luca de Alfaro, Thomas A. Henzinger, and Rupak Majumdar. Symbolic algorithms for infinite-state games. In Kim G. Larsen and Mogens Nielsen, editors, *CONCUR 2001 — Concurrency Theory*, LNCS, pages 536–550, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. doi:10.1007/3-540-44685-0_36.

16 Anne Edmundson, Roya Ensafi, Nick Feamster, and Jennifer Rexford. Nation-state hegemony in internet routing. *Proceedings of the 1st ACM SIGCAS Conference on Computing and Sustainable Societies*, 2018. doi:10.1145/3209811.3211887.

17 S. Enevoldsen, K.G. Larsen, and J. Srba. Abstract dependency graphs and their application to model checking. In *Proceedings of the 25th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'19)*, volume 11427 of *LNCS*, pages 316–333. Springer-Verlag, 2019. doi:10.1007/978-3-030-17462-0_18.

18 Søren Enevoldsen, Kim Guldstrand Larsen, and Jiří Srba. Extended abstract dependency graphs. *International Journal on Software Tools for Technology Transfer*, 24(1):49–65, February 2022. doi:10.1007/s10009-021-00638-8.

19 M. Hendriks, Wang Yi, P. Petterson, J. Hakansson, K.G. Larsen, A. David, G. Behrmann, M. Hendriks, Wang Yi, P. Petterson, J. Hakansson, K.G. Larsen, A. David, and G. Behrmann. Uppaal 4.0. In *Third International Conference on the Quantitative Evaluation of Systems - (QEST'06)*, pages 125–126, 2006. doi:10.1109/QEST.2006.59.

20 Jonas Finnemann Jensen, Kim Guldstrand Larsen, Jiří Srba, and Lars Kaerlund Oestergaard. Efficient model-checking of weighted CTL with upper-bound constraints. *International Journal on Software Tools for Technology Transfer*, 18(4):409–426, August 2016. doi:10.1007/s10009-014-0359-5.

21 Nicolaj Ø. Jensen, Peter G. Jensen, and Kim G. Larsen. Dynamic extrapolation in extended timed automata. In Yi Li and Sofiène Tahar, editors, *Formal Methods and Software Engineering*, LNCS, pages 83–99, Singapore, 2023. Springer Nature Singapore. doi:10.1007/978-981-99-7584-6_6.

22 Nicolaj Ø. Jensen, Kim G. Larsen, and Jiří Srba. Token elimination in model checking of petri nets. In *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS, 2025. doi:10.1007/978-3-031-90643-5_11.

23 Nicolaj Ø. Jensen, Kim G. Larsen, Didier Lime, and Jiri Srba. On-the-fly symbolic algorithm for timed atl with abstractions reproducibility package, April 2025. doi:10.5281/zenodo.15195408.

24 Peter Gjøl Jensen, Kim Guldstrand Larsen, Jiří Srba, and Nikolaj Jensen Ulrik. Elimination of detached regions in dependency graph verification. In Georgiana Caltais and Christian Schilling, editors, *Model Checking Software*, LNCS, pages 163–179, Cham, 2023. Springer Nature Switzerland. doi:10.1007/978-3-031-32157-3_9.

25 Xinxin Liu and Scott A. Smolka. Simple linear-time algorithms for minimal fixed points. In Kim G. Larsen, Sven Skyum, and Glynn Winskel, editors, *Automata, Languages and Programming*, pages 53–66, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg. doi:10.1007/BFb0055040.

26 Oded Maler, Amir Pnueli, and Joseph Sifakis. On the synthesis of discrete controllers for timed systems. In Ernst W. Mayr and Claude Puech, editors, *STACS 95*, LNCS, pages 229–242, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg. doi:10.1007/3-540-59042-0_76.

27 Mark (Mohammad) Tehranipoor, Ujjwal Guin, and Domenic Forte. *Counterfeit Integrated Circuits*, pages 15–36. Springer International Publishing, Cham, 2015. doi:10.1007/978-3-319-11824-6_2.

Optimal Concolic Dynamic Partial Order Reduction

Mohammad Hossein Khoshechin Jorshari
Max Planck Institute for Software Systems (MPI-SWS), Kaiserslautern, Germany

Michalis Kokologiannakis
ETH Zurich, Switzerland

Rupak Majumdar
Max Planck Institute for Software Systems (MPI-SWS), Kaiserslautern, Germany

Srinidhi Nagendra
Max Planck Institute for Software Systems (MPI-SWS), Kaiserslautern, Germany

Abstract

Stateless model checking (SMC) software implementations requires exploring both concurrency- and data nondeterminism. Unfortunately, most SMC algorithms focus on efficient exploration of concurrency nondeterminism, thereby neglecting an important source of bugs.

We present ConDpor, an SMC algorithm for unmodified Java programs that combines optimal dynamic partial order reduction (DPOR) for concurrency nondeterminism, with concolic execution for data nondeterminism. ConDpor is sound, complete, optimal, and parametric w.r.t. the memory consistency model. Our experiments confirm that ConDpor is exponentially faster than DPOR with small-domain enumeration. Overall, ConDpor opens the door for efficient exploration of concurrent programs with data nondeterminism.

2012 ACM Subject Classification Theory of computation → Concurrency; Theory of computation → Verification by model checking

Keywords and phrases Stateless model checking, dynamic symbolic execution

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2025.26

Related Version *Extended Version*: `https://doi.org/10.5281/zenodo.15649583` [44]

Supplementary Material *Software (Source Code)*: `https://github.com/mpi-sws-rse/jmc/tree/old_main`; archived at `swh:1:dir:b3b4445b195ebf730735a84dba1a442816e296a3`

Funding This research was funded in part by the DFG project 389792660 TRR 248–CPEC.

Acknowledgements We thank the anonymous reviewers for their detailed and constructive feedback, and Iason Marmanis for helpful feedback and suggestions.

1 Introduction

Systematic state space exploration for concurrent software involves exploring both *concurrency nondeterminism* and *data nondeterminism*. Concurrency nondeterminism arises out of the possible orders in which threads or processes can be scheduled or messages delivered; data nondeterminism arises out of the possible values that variables may take. Both sources are important in finding bugs.

There are orthogonal techniques to handle each source in the context of dynamic (or stateless) model checking. *Dynamic partial order reduction* (DPOR) [29, 30, 6, 48, 45] handles concurrency nondeterminism efficiently by maintaining scheduling constraints along an execution, and exploring distinct schedules if and only if the corresponding constraints lead to non-equivalent executions. Modern DPOR tools explore the concurrency space optimally [6, 45] and also account for errors caused by compiler/CPU reorderings [8, 5, 48, 56].

© Mohammad Hossein Khoshechin Jorshari, Michalis Kokologiannakis, Rupak Majumdar, and Srinidhi Nagendra;
licensed under Creative Commons License CC-BY 4.0

Symbolic execution handles data nondeterminism by maintaining symbolic constraints on data along an execution, and exploring distinct execution paths if and only if the symbolic constraints along these paths are mutually disjoint. Tools based on *dynamic* symbolic execution (also called concolic testing, CT) [31, 61, 19, 23], maintain the symbolic constraints dynamically, and efficiently manage the backtracking symbolic search.

Despite their many individual successes, few tools handle both concurrency and data nondeterminism *simultaneously and optimally*. Current approaches ignore either partial orderings [59, 19, 23] or data nondeterminism [55, 6, 56, 40, 45] or optimality [58, 60].

In this paper, we present ConDpor, a systematic exploration algorithm that combines DPOR with CT. ConDpor does so by maintaining symbolic constraints in the trace constructed by a DPOR algorithm (specifically, TruSt [45]), and backtracking both on scheduling constraints and on symbolic constraints.

But why tackle both issues at the same time in the first place? At a first glance, data- and concurrency bugs seem orthogonal, as many concurrent programs are otherwise deterministic. However, popular concurrent data structures argue otherwise: algorithms like timestamped stack [26] or elimination-backoff stack [37] use timestamps or random identifiers to achieve better efficiency, and thus fundamentally require reasoning about both concurrency- and data nondeterminism. Moreover, concurrency APIs also often require supporting data nondeterminism, as their primitives may fail with multiple error values, each one leading to a different control-flow path. In a nutshell, handling concurrency and data together amplifies stateless model checking to unbounded state spaces.

An interesting aspect of ConDpor lies in how symbolic constraints are generated and manipulated in the context of DPOR. Indeed, as we later show, ConDpor only needs to know when symbolic values are generated, and when symbolic constraints are evaluated. As such, instead of using a dedicated runtime to manipulate the symbolic path constraint according to the executed instructions, ConDpor offloads the manipulation of symbolic expressions to the underlying programming language (similarly to [20, 63, 60]). Specifically, ConDpor provides dedicated datatypes for symbolic types (available to users) that "carry" the symbolic path constraint of the execution. Arithmetic operations on symbolic variables manipulate the path constraint before performing the respective operation, while logical operations record an event in the current trace at which ConDpor can later backtrack.

ConDpor is sound (i.e., explores no false positives), complete (i.e., explores all program behaviors), optimal (i.e., does not explore traces that only differ in the execution of DPOR-independent instructions or in the model satisfying the symbolic constraints), and parametric in the choice of the memory model (i.e., can find bugs due to compiler/CPU reorderings). As our implementation of ConDpor for concurrent Java programs demonstrates, ConDpor is also efficient: its overhead over standard DPOR is proportional to the amount of data nondeterminism in a given program, and ConDpor is able to fully explore challenging concurrent data structures within seconds.

In summary, we make the following contributions.

- We present ConDpor, a combination of optimal DPOR and concolic execution, and prove it sound, complete, and optimal.
- We implement ConDpor in a tool for programs running on the JVM.
- We empirically show that ConDpor can systematically explore concurrent programs involving both data and concurrency nondeterminism efficiently, with only a small memory overhead.

2 Overview

In this section, we provide a motivating example, followed by an informal description of ConDpor, along with some algorithmic and engineering challenges we faced.

2.1 A Motivating Example

We motivate ConDpor using *timestamped stack* [26], a high-performance data structure where each thread maintains a portion of the stack. When a thread pushes an element, it adds it to a thread-local stack and marks it with a timestamp (e.g., by calling a hardware timer). When a thread pops an element, it goes over the top elements of all stacks, and returns the one with the largest timestamp.

Suppose we want to verify a client that makes n concurrent pushes to the stack and then pops an element:

$$\big(\texttt{push}(1) \;\big\|\; \texttt{push}(2) \;\big\|\; \ldots \;\big\|\; \texttt{push}(n)\big)\,;\texttt{pop}() \qquad \text{(NPUSH+POP)}$$

Enumerative approaches like DPOR would use a shared atomic counter to model timestamps: each call that gets a timestamp will atomically fetch-and-increment the counter. These shared access to the counter make pure DPOR explore $n!$ orderings among the pushes.

Instead, a symbolic approach like CT would construct a symbolic counter value for each thread, only constraining that symbolic values in the same thread are linearly ordered. As the pop() operation compares timestamps to find the largest one, a pure concolic approach would first serialize the pushes and then solve the symbolic constraints to find which push had the largest timestamp, leading to $n! \cdot 2^{n-1}$ executions.

ConDpor combines both DPOR and CT: while it does use a symbolic counter for each thread, ConDpor also leverages DPOR and does not order the pushes, thereby only exploring 2^{n-1} executions (corresponding to solving the symbolic constraints). In fact, if each thread pushed k times the difference becomes even more prominent: ConDpor would still consider 2^{n-1} executions, whereas DPOR would explore $(kn)!/(k!)^n$.

While this example shows a "best case" – we have assumed no further contention – our experiments confirm that ConDpor does outperform purely enumerative approaches for many benchmarks (see §6).

2.2 Making Optimal DPOR Concolic

Let us now describe ConDpor. The core of ConDpor is a backtracking search over program events that explores all executions of a given program (up to equivalence). ConDpor explores exactly one execution from each equivalence class, and initiates no redundant explorations.

2.2.1 Equivalence Partitioning

Our notion of equivalence is a symbolic version of DPOR's underlying equivalence partitioning.[1] For example, assuming an underlying *reads-from* equivalence partitioning [22], ConDpor considers two interleavings equivalent if the reads obtain their values from the same writes, and every symbolic value satisfies the same constraints. As an example, consider

[1] ConDpor extends TruSt [45], which supports both Mazurkiewicz [54] and reads-from equivalence [22].

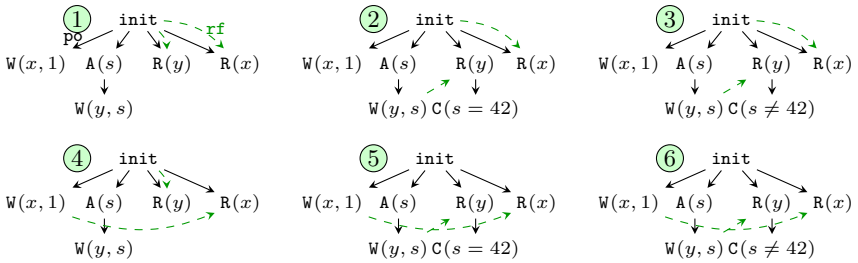


Figure 1 The six execution graphs of W+SW+R+R model its equivalence classes.

the program below[2] where `nondet`() is a source of symbolic values (e.g., program input):

$$x := 1 \;\|\; y := \texttt{nondet}() \;\|\; \texttt{assert}(y = 42) \;\|\; b := x \qquad \text{(W+SW+R+R)}$$

Naively, this program has 36 behaviors. There are $4! = 24$ ways the accesses to x and y can be interleaved. For half of those, the read of y reads 0, whereas for the other half it reads the written symbolic value, which is evaluated in the assertion and may or may not be 42.

Observe, however, that the ordering between the writes to x and y does not matter, as these are different memory locations. As such, it suffices to only consider 6 equivalence classes: two cases for the read of x, and for each of them, one case where the read of y reads 0, and two cases where the read of y reads the written symbolic value (which may or may not be 42).

Formally, the equivalence classes of a given program are represented as a set of *execution graphs* [45]. In each of these graphs, the nodes (events) correspond to instructions of the program, while the edges denote various relations between the instructions. Examples of events are reads and writes to shared variables, while examples of relations include the *program order*, `po`, which orders the instructions of a given thread, and *reads-from*, `rf`, which connects a read to the write from which it obtains its value.

For ConDpor, we extend events to also include the generation of symbolic values and the evaluation of symbolic constraints. For instance, the execution graphs for W+SW+R+R can be seen in Fig. 1. In these graphs, the initializer event `init` models the initialization writes of all memory location (and is `po`-ordered before the first event of each thread), while `R`, `W`, `A` and `C` denote reads, writes, symbolic-value generation and constraint evaluation, respectively. Using these new events means that we can now check for satisfiability of all constraints simply by taking the conjunction of all constraint-evaluation events.

Execution graphs have to satisfy certain consistency constraints imposed by the underlying memory model. For example, under sequential consistency [51], given a program where thread I writes two variables ($x := 1; y := 1$) and thread II reads them in the opposite order ($a := y; b := x$), thread II cannot read $a = 1 \wedge b = 0$. Other models (known as weak memory models) allow this outcome, which may arise due to compiler and/or CPU optimizations.

Although below we assume sequential consistency to ease the presentation, ConDpor is parametric in the choice of the memory consistency model (see §3).

[2] We use x, y, z for shared variables (initialized to 0), and $a, b, \ldots$ for thread-local variables.

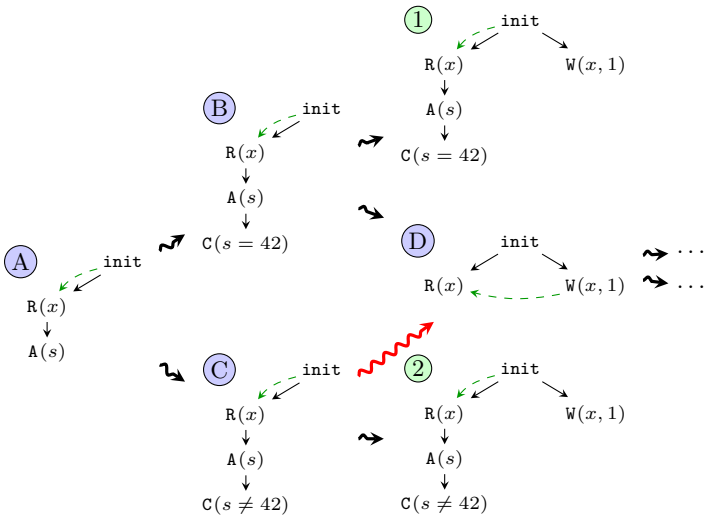


Figure 2 A ConDpor exploration.

2.2.2 Enumerating Execution Graphs

Given the representation above, we can verify a concurrent program by enumerating all of its execution graphs. ConDpor does so in a depth-first manner: starting from the empty graph, it extends it one event at a time (maintaining consistency), recording alternative exploration options when possible. When a read is added to the graph, ConDpor explores all writes the read can (consistently) read from, and sets its `rf` accordingly. When a write is added, ConDpor checks to see whether any of the previously added reads can be (consistently) revisited to read from the newly added write. Finally, when a constraint-evaluation event is added, ConDpor examines all both outcomes, assuming both are satisfiable.

▶ **Example 1.** Let us demonstrate how ConDpor works with an example.

$$\begin{array}{l} a := x \\ b := \texttt{nondet}() \\ \texttt{assert}(a = 0 \vee b \neq 42) \end{array} \;\Big\|\; x := 1 \qquad \text{(RS+W)}$$

In the program above, an error manifests if thread I both reads 1 for x, and `nondet()` returns the unexpected value 42. ConDpor's exploration for RS+W is depicted in Fig. 2.

At the first step, ConDpor adds $\texttt{R}(x)$ to the graph; this read can only read 0, as there is only one write to x in the current graph (the `init` event). When ConDpor encounters `nondet()`, it adds an $\texttt{A}(s)$ label to the graph. This label is used to keep track of the domain of the symbolic value (see §3), and also to inform the SMT solver of the new variable.

When ConDpor reaches the assert statement, while the value of a is already known, the result of $b \neq 42$ is not. As such, ConDpor examines both options: in graph (B), we assume that $b = 42$ (since the SMT solver returns a satisfying assignment for $s = 42$), while in graph (C) we assume that $b \neq 42$ (again, the SMT solver is queried[3] for the satisfiability of $s \neq 42$).

[3] As in concolic execution, one of these two calls can be spared; see §4.

In graphs (B) and (C), ConDpor subsequently encounters the statement $x := 1$. Merely adding the respective write event to the graph, however, is inadequate, as ConDpor also has to explore the case where thread I reads from this write. As such, ConDpor also examines whether $\mathtt{W}(x, 1)$ can revisit any of the existing same-location reads in the graph.

Revisiting complicates the exploration, as ConDpor has to *restrict* graphs resulting from revisits in order to ensure consistency. Indeed, in our example above ConDpor removes all $(\mathtt{po} \cup \mathtt{rf})^+$-successors of $\mathtt{R}(x)$ in the resulting graph (D), as the existence of these events might be (control-flow) dependent on the read value.

Also observe that we need to maintain *optimality*: each execution graph should be explored exactly once. In the exploration above, if we are not careful and revisit $\mathtt{R}(x)$ from both (B) and (C), we will get graph (D) twice, leading to duplication. ConDpor ensures $\mathtt{R}(x)$ is only revisited in exploration (B).

Optimality

ConDpor ensures optimality by extending the notion of maximal extensions [45] from existing graph-based DPOR algorithms so that it accounts for constraint evaluation events. The key idea behind maximal extensions is roughly that if performing the same revisit in two graphs G_1 and G_2 leads to the same graph, then G_1 and G_2 only differ in the events deleted by the revisit. In the example above, graphs (B) and (C) differ in their constraint evaluation event. In turn, by imposing a criterion on the events deleted by the revisit – that they must be maximally added – we can perform this in only one of the possible sub-explorations.

We defer the presentation of our maximal-extension definition to §4, where we also show how they can be efficiently computed.

2.3 Engineering Challenges

ConDpor works for any programming language, assuming that concurrency primitives (e.g., shared-memory reads and writes) can be intercepted, and that symbolic expressions can be manipulated along program execution (e.g., with a symbolic interpreter).

2.3.1 Intercepting Concurrency Primitives in the JVM

DPOR implementations intercept concurrency primitives through system call interception [55], through a custom runtime [48, 35], or through mocking concurrency libraries [64, 27].

Unfortunately, neither technique is appropriate for the JVM: distinguishing system calls from the program under test from system calls made by the JVM itself is difficult, writing and maintaining a custom runtime for Java is a multi-person-year effort, and mocking is inapplicable because some concurrency primitives are handled natively within the JVM.

Our solution is to instrument the program through bytecode rewriting using the ASM library [1], and to provide our own asynchronous runtime for concurrency operations. Our instrumentation provides additional synchronization "under the hood" yielding the scheduler full control over Java's concurrency constructs.

2.3.2 Symbolic Execution without an Interpreter

As with concurrency primitives, manipulating symbolic expressions typically requires a custom interpreter for the instructions (e.g., [19], [15]). Sometimes, the interpreter instructions are instrumented into the code so that the symbolic interpretation occurs as the program runs.

While we do perform bytecode rewriting, the cost of developing a full symbolic interpreter for the JVM is quite high, and the interpreter has to understand and differentiate between the code under test and JVM's own operations.

To overcome this issue, we use Java's type system to push the symbolic interpretation into the language itself. More specifically, in addition to all constructs made available to the user by the language, ConDpor provides an API defining symbolic types (e.g., symbolic integers). These types provide the same operations as their non-symbolic counterparts, but also "carry" a symbolic expression that is being manipulated along with the respective values.

To manipulate symbolic values, the API provides the operations `nondet`() and `evaluate`(), which create a symbolic value and evaluate a symbolic constraint, respectively. Note that the purpose of evaluation is twofold: apart from allowing the result of a symbolic expression to be used in non-symbolic contexts (e.g., in the condition of an if-statement), it also serves as a backtracking point for ConDpor. Indeed, whenever `evaluate`() is encountered, the runtime calls ConDpor-specific code to add an `C` event to the graph (see § 2.2). While having such an API means that users must use it to capture all sources of data nondeterminism, such a constraint can be easily lifted (e.g., by means of a data-flow analysis over bytecode).

Crucially, our symbolic API remains a part of the program under test: it is implemented fully in Java and requires no special runtime. Internally, our API calls ConDpor-specific code only to register the generation/evaluation of a symbolic expression.

3 Partial Order Semantics

We now formally describe execution graphs, the data structure behind ConDpor's partial order semantics. We begin by defining events.

▶ **Definition 2.** *An event, $e \in \mathsf{Event}$, is either the initialization event* `init` *or a thread event $\langle t, i, lab\rangle$, where $t \in \mathsf{Tid}$ is a thread identifier, $i \in \mathsf{Idx} \triangleq \mathbb{N}$ an index within a thread, and $lab \in \mathsf{Lab}$ a label that takes one of the following forms:*

- *Write label, $\mathtt{W}(l, v)$, where $l \in \mathsf{Loc}$ is the location accessed and $v \in \mathsf{Val}$ is the value written.*
- *Read label, $\mathtt{R}(l)$, where $l \in \mathsf{Loc}$ is the location accessed.*
- *Symbolic generation label, $\mathtt{A}(s)$, where $s \in \mathsf{Sexpr}$ is the symbolic value generated.*
- *Constraint evaluation label, $\mathtt{C}(\phi)$, where $\phi \in \mathsf{Sexpr}$ is a boolean symbolic expression.*
- *Block label, `B`, denoting thread blockage (see below).*
- *Error label,* `error`*, denoting a program error (see below).*

We define the set of all read events as $\mathtt{R} \triangleq \{\langle t, i, lab\rangle \mid lab = \mathtt{R}(_)\}$. Similarly, we denote the set of all write, symbolic-generation, and constraint-evaluation events as `W`*,* `A`*, and* `C`*, respectively, and assume that* `init` $\in$ `W`*.*

Block and error labels are generated by `assume` and `assert` statements, respectively. An `assume`(b) statement is modeled as `if` ($!b$) `block()`, while an `assert`(b) statement is modeled as `if` ($!b$) `error()`. The `block()` and `error()` statements generate the respective labels, while $b \in \mathsf{Val}$ is a boolean value.

Having defined events, we define execution graphs as follows.

▶ **Definition 3.** *An execution graph G consists of:*

(1) *a set of events E that includes* `init` *and does not contain multiple events with the same thread identifier and serial number;*

(2) *the* reads-from *function* `rf` $: E \cap \mathtt{R} \to E \cap \mathtt{W}$ *that maps each read to the same-location write event from which it gets its value;*

(3) *the* coherence order $\mathtt{co} \subseteq \bigcup_{l \in \mathsf{Loc}} \mathtt{W}_l \times \mathtt{W}_l$ *(with* $\mathtt{W}_l \triangleq \{\langle t, i, lab\rangle \in \mathtt{W} \mid lab = \mathtt{W}(l, _)\}$*), a strict partial order that is total on* $\mathtt{W}_l$ *for every location* $l \in \mathsf{Loc}$*; and*

(4) *a total order* $\leq$ *on* E*, representing the order in which events were added to the graph.*

We write $G.\mathtt{E}$, $G.\mathtt{rf}$, $G.\mathtt{co}$, and $\leq_G$ to project the various components of an execution graph, and use $G|_E$ to denote the restriction of an execution graph G to a set of events E. We assume that $\mathtt{init} \in \mathtt{W}$, and use the functions $\mathtt{tid}$, $\mathtt{idx}$, $\mathtt{loc}$, $\mathtt{val}$ and $\mathtt{formula}$ to get (when applicable) the thread identifier, index, location, value and formula of an event, respectively. We write $G.\Phi \triangleq \bigwedge_{s \in G.\mathtt{C}} \mathtt{formula}(s)$ for the conjunction of all symbolic constraints of the graph, $G.\mathtt{W}$ for $G.\mathtt{E} \cap \mathtt{W}$ (and similarly for other sets), and use subscripts to further restrict these sets (e.g., $\mathtt{W}_l \triangleq \{w \in \mathtt{W} \mid \mathtt{loc}(w) = l\}$). Finally, given two events $e_1, e_2 \in G.\mathtt{E}$, we write $e_1 <_G e_2$ if $e_1 \leq_G e_2$ and $e_1 \neq e_2$.

As defined above, execution graphs do not have an explicit *program order* ($\mathtt{po}$) component. We thus induce $\mathtt{po}$ based on our event representation as follows:

$$\mathtt{po} \triangleq \{\langle \mathtt{init}, e\rangle \mid e \in \mathsf{Event} \setminus \{\mathtt{init}\}\} \cup \{\langle\langle t_1, i_1, lab_1\rangle, \langle t_2, i_2, lab_2\rangle\rangle \mid t_1 = t_2 \wedge i_1 < i_2\}$$

We write $G.\mathtt{porf} \triangleq (\mathtt{po} \cap (G.\mathtt{E} \times G.\mathtt{E}) \cup \{\langle G.\mathtt{rf}(r), r\rangle \mid r \in G.\mathtt{R}\})^+$ for the graph's causal dependency relation.

Finally, we define the semantics of a program P under a model M as the set of execution graphs corresponding to P that satisfy M's consistency predicate. In this paper, we assume a memory model M providing a consistency predicate $\mathsf{consistent}_{\mathsf{M}}(G)$, determining whether a graph G is consistent, as well as an error predicate $\textsc{IsErroneous}_{\mathsf{M}}(G)$, determining whether G contains an error (e.g., a data race) according to M. We further assume that $\mathsf{consistent}_{\mathsf{M}}(\cdot)$ is extensible, prefix-closed, and implies RMW atomicity and $\mathtt{porf}$ acyclicity (see [45]), and that $\textsc{IsErroneous}_{\mathsf{M}}(\cdot)$ is monotone (if a prefix-closed subset of a graph G contains an error, then so does G). Models satisfying these assumptions include SC [51], TSO [57] and RC11 [50] under shared-memory consistency, as well as asynchronous communication, peer-to-peer, and mailbox under message-passing consistency [25].

Given such an underlying memory consistency model M, we then define a new memory model M_S, with $\mathsf{consistent}_{\mathsf{M}_S}(G) \triangleq \mathsf{consistent}_{\mathsf{M}}(G) \wedge \mathtt{SAT}(G.\Phi)$ and $\textsc{IsErroneous}_{\mathsf{M}_S}(G) \triangleq \textsc{IsErroneous}_{\mathsf{M}}(G)$, where $\mathtt{SAT}(\cdot)$ is a predicate denoting whether a symbolic constraint is satisfiable. Intuitively, M_S extends the underlying communication semantics to also require that all symbolic constraints in a graph are satisfiable. It is easy to show that M_S satisfies the memory-model properties stated above.

4 Algorithm

Let us now present ConDpor in detail (cf. Algorithm 1). Although our algorithm works both under the Shasha-Snir and the reads-from equivalence partitioning, here we only provide the (more straightforward) Shasha-Snir version [62], the generalization of Mazurkiewicz equivalence partitioning for relaxed memory models such as TSO and PSO. Symbolic reasoning does not affect the choice of partitioning, and the required changes can be adapted from [45].

Given a program P, ConDpor explores all of its consistent execution graphs under a model M by calling $\textsc{Visit}_{P,\mathsf{M}_S}(G_\emptyset)$, where $G_\emptyset$ is the initial graph containing only the initialization event $\mathtt{init}$. In turn, Visit constructs the execution graphs of P one at a time and incrementally, while also recording the event addition order in the graph's $\leq$ component.

Let us now examine Visit more closely. As a first step, Visit checks whether G contains an error (line 2); errors include safety violations (which manifest with an $\mathtt{error}$ label), as well as memory-model-specific errors such as data races.

Algorithm 1 ConDpor: Optimal Concolic Dynamic Partial Order Reduction.

```
1:  procedure Visit_{P,M_S}(G)
2:      if IsErroneous_{M_S}(G) then exit("Erroneous execution")
3:      a ← Add(G, next_P(G))
4:      if a = ⊥ then return "Visited full execution graph G"
5:      switch a do
6:          case a ∈ R
7:              for w ∈ G.W_{loc(a)} do VisitIfConsistent_{P,M_S}(SetRF(G, a, w))
8:          case a ∈ C
9:              VisitIfConsistent_{P,M_S}(SetSYM(G, a, formula(a)))
10:             VisitIfConsistent_{P,M_S}(SetSYM(G, a, ¬formula(a)))
11:         case a ∈ W
12:             VisitCOs_{P,M_S}(G, a)
13:             for r ∈ G.R_a such that ⟨r, a⟩ ∉ G.porf do
14:                 MaybeBackwardRevisit_{P,M_S}(G, r, a)
15:         case _ Visit_{P,M_S}(G)
16: procedure VisitCOs_{P,M_S}(G, a)
17:     for w_p ∈ G.W_{loc(a)} do VisitIfConsistent_{P,M_S}(SetCO(G, w_p, a))
18: procedure MaybeBackwardRevisit_P(G, r, a)
19:     Deleted ← {e ∈ G.E | r <_G e ∧ ⟨e, a⟩ ∉ G.porf}
20:     [d_1, ..., d_n] ← sort_{<_G}(Deleted)
21:     if ∃G', G'' such that G' ⇝^r G'' ⇝^{d_1} ··· ⇝^{d_n} G|_{G.E\{a}} then
22:         VisitIfConsistent_{P,M_S}(SetRF(G|_{G.E\Deleted}, r, a))
23: procedure VisitIfConsistent_{P,M_S}(G)
24:     if consistent_{M_S}(G) then Visit_{P,M_S}(G)
```

After ensuring that the graph is error-free, Visit extends the current graph with the next program event (line 3) with the help of Add and next. next returns an event from a thread that is not blocked or finished, and Add adds it to the graph in place, and then returns the new event. If there are no events to add (in which case Add/next returns $\bot$), a full execution of P has been explored, and Visit returns (line 4).

If a is a read, we recursively explore all consistent rf options for it (line 7). (We assume that $\mathsf{SetRF}(G, r, w)$ returns a new graph G' that only differs from G in rf: $G'.\mathtt{rf}(r) = w$.)

If a is a constraint-evaluation event, ConDpor has to examine whether the newly added event (and its negation) renders $G.\Phi$ satisfiable. To do that, Visit calls VisitIfConsistent twice: once with a's constraint as is, and another time with a's expression negated. In order to set a's symbolic constraint, Visit employs the $\mathsf{SetSYM}(G, a, \phi)$ function, which returns a new graph G' that is identical to G, apart from the constraint of event a, which is set to ϕ. Observe that if $G'.\Phi$ is unsatisfiable, then the execution subsequently be dropped by VisitIfConsistent as inconsistent.

If a is a write, ConDpor examines both the non-revisit- and the revisit case. In the non-revisit case, ConDpor explores all consistent co options for a via VisitCOs (line 12). Analogously to SetRF, $\mathsf{SetCO}(G, w_p, w)$ returns a new graph G' that only differs from G in that the co-predecessor of w is w_p.

In the revisit case, Visit examines previously-added reads as revisit candidates. Concretely, Visit only revisits same-location reads that are not porf-before a (as revisiting those would create porf cycles), assuming that the events deleted from the revisit have been added maximally (lines 13-14). The maximal-addition definition closely follows the one of

Kokologiannakis et al. [46]: ConDpor backward-revisits r from a if all deleted events are added co-maximally (if non-symbolic), or in a unique fashion (if symbolic). Formally, we write $G_1 \overset{e}{\rightsquigarrow} G_2$ if $G_2 = \textsc{Add}(G_1, e)$ and:

$$
\begin{array}{llll}
G_2.\mathtt{rf} = G_1.\mathtt{rf} \cup \{\langle \mathsf{max}_{G_1.\mathtt{co}_e}, e\rangle\} & G_2.\mathtt{co} = G_1.\mathtt{co} & G_2.\Phi = G_1.\Phi & \text{if } e \in \mathtt{R} \\
G_2.\mathtt{rf} = G_1.\mathtt{rf} & G_2.\mathtt{co} = G_1.\mathtt{co} \cup (G_1.\mathtt{W}_e \times \{e\}) & G_2.\Phi = G_1.\Phi & \text{if } e \in \mathtt{W} \\
G_2.\mathtt{rf} = G_1.\mathtt{rf} & G_2.\mathtt{co} = G_1.\mathtt{co} & G_2.\Phi = \mathtt{ST}(G_1.\Phi, e) & \text{if } e \in \mathtt{C} \\
G_2.\mathtt{rf} = G_1.\mathtt{rf} & G_2.\mathtt{co} = G_1.\mathtt{co} & G_2.\Phi = G_1.\Phi & \text{otherwise}
\end{array}
$$

Above, the $\mathtt{ST}(\cdot)$ function acts as a tiebraker if both cases of a constraint-evaluation event are feasible. If both $\mathtt{formula}(e)$ and $\neg\mathtt{formula}(e)$ are feasible, it (arbitrarily) returns $G_1.\Phi \wedge \mathtt{formula}(e)$, while if only $\mathtt{formula}(e)$ (resp. $\neg\mathtt{formula}(e)$) is feasible, it returns $G_1.\Phi \wedge \mathtt{formula}(e)$ (resp. $G_1.\Phi \wedge \neg\mathtt{formula}(e)$).

4.1 Optimizing the Algorithm

Observe that Algorithm 1 performs a lot of redundant calls to the SMT solver. As one example, checking graph consistency requires checking the satisfiability of $G.\Phi$. However, as $G.\Phi$ only changes when adding evaluation labels, it suffices to only call the solver in the respective cases. As another example, observe that satisfiability is also necessary in the $\mathtt{ST}(\cdot)$ function, when checking for maximality of symbolic evaluation events. We can spare these solver calls by leveraging concolic execution, as explained below.

Indeed, observe that Algorithm 1 does not immediately leverage concolic execution: the SMT solver is called twice when adding a symbolic constraint (lines 9-10), as well as to evaluate the maximality of constraint-evaluation events during backward revisits ($\mathtt{ST}(\cdot)$ function). As done in concolic execution, however, if we make our symbolic types "carry" a model of each symbolic value along with the respective symbolic expression, one of these two calls can be spared. Concretely, when adding a constraint-evaluation event, we can evaluate the non-negated constraint (line 9) without resulting to an SMT solver (by using the model).

Backward revisits can also leverage concolic execution and spare SMT calls in the $\mathtt{ST}(\cdot)$ function, though with a bit more care. Assuming G_r is the graph resulting from the revisit, we first check G_r's consistency[4] and obtain a model of all symbolic values in G_r. Then, we iterate over the events in the deleted set (lines 19-20), and consider a constraint-evaluation event a as maximal, if $\mathtt{formula}(a)$ is true in the model obtained in G_r. If that's not the case, then this means that $\neg\mathtt{formula}(a)$ is feasible, and hence that case is deemed as maximal. (Any symbolic generation events encountered, simply extend the existing model.)

Note, however, that the above procedure requires the model that is returned by an SMT solver for a given formula to be deterministic (which is the case with e.g., Z3). To see why, consider the following program, where a symbolic value is shared between two threads:

$$
\begin{array}{ccccc}
 & & a := \mathtt{nondet}() & & \\
b := x & \Big\| & \mathtt{assume}(a > 42) & \Big\| & \mathtt{if}\ (a > 0)\ \ x := 1
\end{array}
$$

There are two executions in which $x := 1$ can revisit $b := x$: one where thread II has a $\mathtt{C}(a > 42)$ event (and the assume succeeds) and one where it has a $\mathtt{C}(a \leq 42)$ event (and the assume blocks). In both cases, the graph occurring if $\mathtt{W}(x, 1)$ revisits $\mathtt{R}(x)$ will contain $\mathtt{R}(x)$, $\mathtt{C}(a > 0)$ and $\mathtt{W}(x, 1)$, while the deleted set will be $\mathtt{C}(a > 42)$ and $\mathtt{C}(a \leq 42)$ respectively. If,

[4] VisitIfConsistent in line 15 can reuse this result.

however, the model we obtain from the solver for $\mathtt{C}(a > 0)$ is $a = 43$ in the first case and $a = 1$ in the second case, then the revisit will not occur, as each case would drop it, assuming it would take place in the other one.

4.2 Soundness, Completeness and Optimality

Given a program P and a consistency model M that satisfies the conditions of §3, Algorithm 1 is sound (i.e., only explores consistent full executions), complete (i.e., explores all of P's consistent executions), and optimal (i.e., explores each consistent execution exactly once, and does not engage in any wasteful explorations). The proofs follow by extending the framework of [46] to account for symbolic event types.

Before formally stating our results, let us provide some definitions.

- We write $[\![P]\!]_{\mathsf{M}_S}$ for the set of consistent full graphs corresponding to the program P under M_S, and $G_1 \approx G_2$ if G_1 and G_2 agree on all their components but $\leq_G$.
- Given graphs G_1, G_2, we say that G_1 is a *prefix* of G_2 (written $G_1 \sqsubseteq G_2$), if there exists a graph $G_1' \approx G_1$ such that the algorithm can reach a graph $G_2' \approx G_2$ from G_1' in a series of non-revisit steps.

Let us now state our results. Observe that ConDpor is trivially sound, as Algorithm 1 never visits inconsistent executions.

▶ **Theorem 4** (Completeness). *Let G_f be an execution graph in $[\![P]\!]_M$. Then $\textsc{Visit}_{P,\mathsf{M}_S}(G_\emptyset)$ calls $\textsc{Visit}_{P,\mathsf{M}_S}(G_f')$ for some $G_f' \approx G_f$.*

▶ **Theorem 5** (Optimality). *$\textsc{Visit}_{P,\mathsf{M}_S}(G_\emptyset)$ never calls $\textsc{Visit}_{P,\mathsf{M}_S}$*
(a) *for graphs G_1, G_2 such that $G_1 \approx G_2$, or*
(b) *for a graph G that cannot lead to a full execution $G_f \in [\![P]\!]_{\mathsf{M}_S}$.*

We prove both theorems by showing the exact (unique) sequence of steps that the algorithm takes to reach a graph G_f, with $G_f \approx G_f' \in [\![P]\!]_{\mathsf{M}_S}$.

Given $G_s \sqsubseteq G_t$, we define a procedure $\textsc{GetNext}(G_s, G_t)$ that returns the (minimal, unique and nonempty) sequence of algorithmic steps S with which G_s reaches a graph G such that $G_s \sqsubseteq G \sqsubseteq G_t$, and with the property (proved by induction) that S does not revisit events in G_s (it may revisit events not in G_s). Crucially, the inductive proof for $\textsc{GetNext}(G_s, G_t)$ relies on the ability to extend its first argument G_s, which is not affected by our new symbolic events (M_S is maximally extensible).

Calling GetNext in a fixpoint yields the desired result.

5 Implementation

We implemented ConDpor as a tool for concurrent Java programs[5]. Our tool supports commonly used Java synchronization primitives such as locks, monitors, synchronized blocks, as well as thread creation and joining.

ConDpor can operate in two modes: a verification mode, and a "random mode" in which ConDpor samples executions from the program state space. The latter mode is useful for finding bugs faster, as random sampling does not involve backtracking.

Verifying a program with ConDpor involves three steps:
(1) compiling it into Java bytecode,
(2) instrumenting the bytecode so that we can intercept operations of interest, and

[5] https://github.com/mpi-sws-rse/jmc

(3) running the instrumented bytecode under a special runtime that implements ConDpor to systematically explore all program behaviors.

Even though our algorithm is parametric in the choice of the memory model, our implementation currently only works under SC [51].

To instrument the bytecode, we used the ASM library [1], and inserted yield calls to ConDpor at points of interest. For example, whenever a thread acquires or releases a lock, the instrumented bytecode yields control to ConDpor, which then updates the execution graph and schedules threads appropriately. ConDpor controls thread scheduling by employing a shared mutex between the native Java Thread and the scheduler thread. We implement the `yield` method as `Object.wait` over this shared mutex, effectively imposing a cooperative multithreading semantics on top of the Java runtime.

To return appropriate values during constraint evaluation, ConDpor uses the JavaSMT library [13] to query an SMT solver for satisfiability. Although JavaSMT can interface with multiple solvers, we have only experimented with `Z3` [3]. Similarly to TruSt [45], ConDpor copies the graph whenever a write revisits a read, meaning that a new solver instance has to be created as well. We found that creating new instances is expensive, and implemented a shared solver pool with a fixed number of instantiated solvers to allow solver reuse. Each solver instance also leverages incremental solving, which turned out to be very beneficial.

Finally, note that users must utilize our symbolic API to capture all sources of data nondeterminism manually: in practice, this involves subclassing some input types to their symbolic counterparts. Note that some existing symbolic (or concolic) execution engines, like EXE [20], employ source-to-source translation, where the instrumentation is written in the same programming language as the program under test. While a source-to-source translation is also possible for Java, we decided to implement ConDpor using bytecode instrumentation and by encoding symbolic objects directly in the language, so as to avoid maintaining a Java compiler infrastructure.

6 Evaluation

We now proceed with ConDpor's evaluation, which aims to answer the following questions:

§ 6.1 How does symbolic handling of data nondeterminism fare against an explicit one?

§ 6.2 Can ConDpor handle real-world programs that employ data nondeterminism?

To answer these questions, we conduct two case studies. To showcase the differences between explicit and symbolic handling of data nondeterminism, we evaluate ConDpor on a set of synthetic benchmarks designed to juxtapose the two approaches. To see how well ConDpor scales in realistic code, we use a diverse set of concurrent data structures that rely on randomness (modeled as data nondeterminism). In both studies, we compare ConDpor against a baseline DPOR implementation ExEn that handles data nondeterminism explicitly by exhaustively enumerating all possible values in the input domain, and repeatedly running vanilla DPOR for each value. To ensure a fair comparison, we restrict the data domain to $\{0, 1, 2\}$, making explicit enumeration feasible.

Our evaluation demonstrates that ConDpor's handling of nondeterminism is exponentially better than ExEn's, and that ConDpor is able to completely explore clients of data-nondeterministic concurrent data structures. Moreover, ConDpor's overhead over vanilla DPOR is proportional to the amount of data nondeterminism in the input program.

Table 1 ConDpor performs better than ExEn even for small data domains.

	ExEn			ConDpor		
	Execs	*Time*	*Mem*	*Execs*	*Time*	*Mem*
`SVQueue1(8)`	39 366	365	73	4	0	77
`SVQueue1(9)`	118 098	1743	73	4	0	71
`SVQueue1(10)`	🕐	🕐	🕐	4	0	70
`SVQueue2(2)`	1242	5	79	138	1	70
`SVQueue2(3)`	92 556	398	108	3428	13	100
`SVQueue2(4)`	🕐	🕐	🕐	86 708	530	118
`SVQueue3(7)`	🕐	🕐	🕐	2865	27	100
`SVQueue3(8)`	🕐	🕐	🕐	6506	70	130
`SVQueue3(9)`	🕐	🕐	🕐	14 505	175	158
`SVStack1(5)`	183 222	1481	126	754	5	108
`SVStack1(6)`	🕐	🕐	🕐	2770	26	131
`SVStack1(7)`	🕐	🕐	🕐	10 294	112	148
`SVStack2(6)`	🕐	🕐	🕐	1481	14	123
`SVStack2(7)`	🕐	🕐	🕐	4876	54	153
`SVStack2(8)`	🕐	🕐	🕐	16 422	213	172
`Counter1(4)`	11 988	32	70	2368	9	78
`Counter1(5)`	317 115	1044	100	41 760	217	90
`Counter1(6)`	🕐	🕐	🕐	883 584	5540	200
`Counter2(5)`	66 465	215	180	5770	25	140
`Counter2(6)`	1 411 770	6368	200	69 852	403	155
`Counter2(7)`	🕐	🕐	🕐	970 886	6929	220

Experimental Setup

We conducted all experiments on a Dell Latitude 5450 system running a custom Debian-based distribution with an Intel Core Ultra 5 135H CPU (18 cores @ 4.60 GHz) and 32GB of RAM.

In all tables, *Execs* denotes the number of executions explored, *Time* the required time (in seconds), and *Mem* the required memory (in MB). We set a timeout of 210 minutes and a memory limit of 512MB (denoted by 🕐 and OOM).

6.1 Explicit vs Symbolic Data Nondeterminism

Let us begin by comparing ConDpor to ExEn on synthetic benchmarks. For this part of our evaluation, we used all benchmarks employing data nondeterminism from SV-COMP's `pthread` category [67], as well as two handcrafted ones. Our results can be seen in Table 1.

The main takeaway from this table is that, despite the small data domain, for each benchmark there is an input parameter (the size of shared buffer in SV-COMP benchmarks and the number of threads in the handcrafted ones) for which ExEn times out, whereas ConDpor scales beyond it. Moreover the memory increase over ExEn (which consumes polynomial memory) is negligible due to the tests having only a few constraints.

Let us now examine the benchmarks of Table 1 more carefully to gain a better understanding of the differences between the two approaches. Starting with `SVQueue*(N)` and `SVStack*(N)`, these benchmarks implement a simple data structure (queue and stack, respectively) using an array of size N, while a producer and a consumer thread add/remove N symbolic integers using a shared lock. (Queue/stack tests only differ in the way the threads synchronize when modifying the data structure.) As can be seen, ExEn cannot

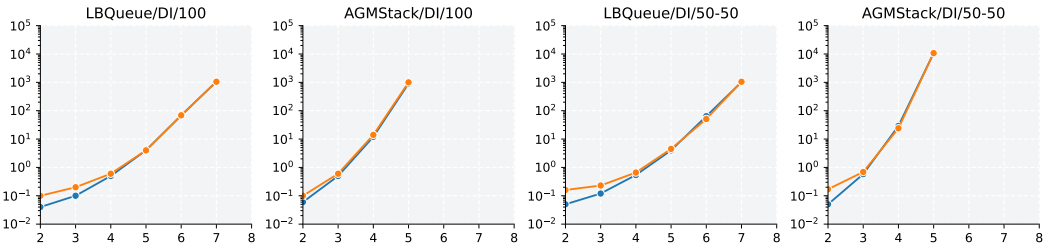


Figure 3 ConDpor adds no overhead over DPOR on data-independent tests (time/threads).

scale beyond very small arrays since, in addition to all possible schedules, it also explicitly enumerates all values for each array cell. ConDpor, on the other hand, scales nicely even for larger arrays, as it only examines whether a removed item matches a previously inserted one. We can draw similar conclusions for the `Counter*(N)` benchmarks, where each of N threads nondeterministically operates on one of two shared counters. Although there is a single source of data nondeterminism in each thread, ExEn quickly times out for larger Ns.

6.2 Verifying Concurrent Data-Structures

Let us now move on to applying ConDpor to realistic concurrent data structures. We classify these benchmarks into three categories, depending on their amount of data nondeterminism.

- *data-independent data structures* (lock-based queue [38] and Afek-Gafni-Morrison Stack [10]). These do not perform any computation on the arguments of their methods.
- *data-dependent data structures* (coarse-grained list [38], fine-grained list [14], optimistic list [38] and lazy list [36]). These check equality or ordering of their method arguments.
- *nondeterminism-based data structures* (timestamped stack [26] and elimination backoff stack [37]). These are data-independent on their arguments, but internally rely on data nondeterminism for improved performance.

We ran ConDpor on each category under two different workloads: a "100-0" workload, where N threads are inserting a symbolic integer to the data structure, and a "50-50" workload where $\lceil \frac{N}{2} \rceil$ threads insert an item and $\lfloor \frac{N}{2} \rfloor$ threads remove an item. (Tables including all experimental results against ExEn can be found in the appendix of our extended version [44].)

6.2.1 Data-Independent Data Structures

We begin by measuring ConDpor's overhead over vanilla DPOR on some data-independent data structures (cf. Fig. 3). As these data structures do not use the values of their arguments, vanilla DPOR and ConDpor explore the same number of executions. What these benchmarks demonstrate, however, is that manipulating symbolic integers in ConDpor does not induce any overhead, as no calls to the SMT solver are performed thanks to concolic execution. Indeed, ConDpor randomly concretizes all nondeterministic values, and never encounters constraint evaluation nodes, allowing it to scale in exactly the same manner as vanilla DPOR. (The small overhead of ConDpor for small input parameters is due to the solver initialization; see §5.)

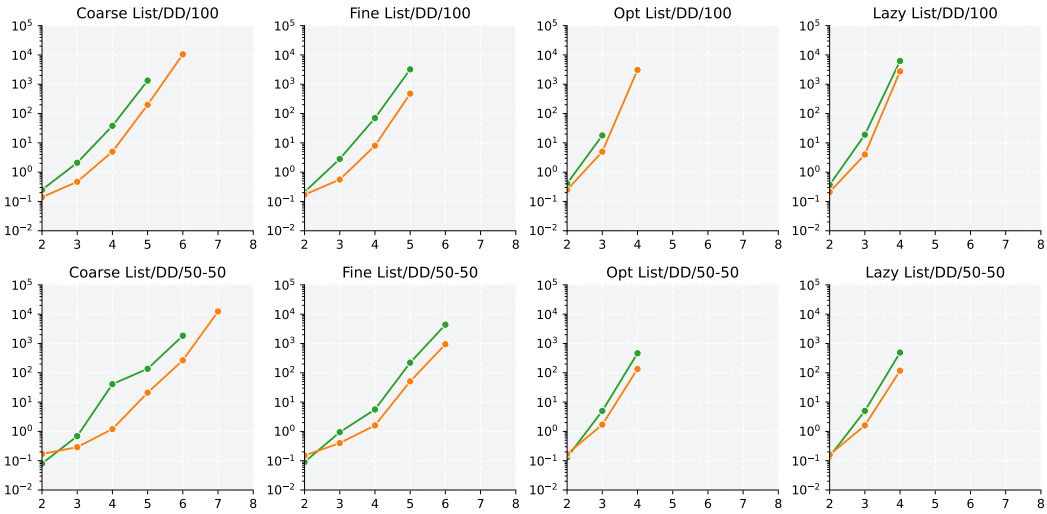


Figure 4 ConDpor scales better than ExEn on data-dependent tests (time/threads).

6.2.2 Data-dependent Data Structures

Data-dependent data structures, on the other hand, do incur some overhead over concurrency nondeterminism. A comparison between ConDpor and ExEn on such structures can be seen in Fig. 4. As expected, ConDpor scales better than ExEn in all benchmarks, and often scales to more threads than ExEn (e.g., in `Coarse List/DD/50-50`).

What we also observe in these benchmarks, however, is that most time is not spent on evaluating symbolic constraints, but rather on enumerating executions (see Appendix A.3 of [44]).

Indeed, taking `Coarse List/DD/100` as an example, ConDpor spends considerably less time evaluating constraints than enumerating executions: for 6 threads, only 12.5% of the total time was spent on solving constraints [44], despite the fact that there are $N!$ ways the symbolic constraints can be evaluated in each of the $N!$ concurrent behaviors. This is because the time per execution is greater than the time required to solve a single constraint. We also observe that ConDpor scales better for the "50-50" workload than "100-0" workload, as in the former ConDpor does not have to evaluate constraints if the data structure is empty.

6.2.3 Nondeterminism-based Data Structures

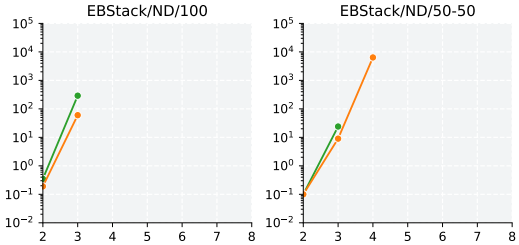


Figure 5 ConDpor outperforms ExEn for data structures employing randomness (time/threads).

The last class of benchmarks discusses data structures that exploit nondeterminism at the core of their operations. Such structures are interesting because although the data domain might be small (e.g., if we use a small elimination array in an elimination structure), ConDpor still provides significant benefits over ExEn or vanilla DPOR (assuming an equivalent encoding is possible).

Similar to the data-dependent data structures, `EBStack/ND/50-50`, as depicted in Figure 5, this confirms the scalability of ConDpor as the input parameter increases compared to ExEn. For instance, in `EBStack` with an elimination array of size $\lceil \frac{N}{2} \rceil$ for "50-50" workload, ExEn still enumerates all values in the data domain, even in the case where threads do not touch the elimination array; by contrast, ConDpor does not evaluate any constraints.

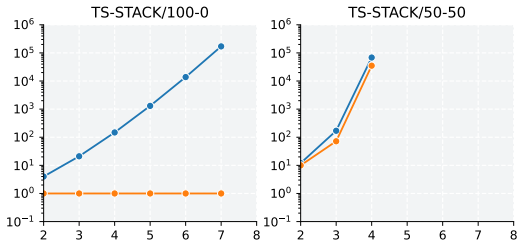


Figure 6 Vanilla DPOR quickly times out when the parameter of `TSStack/100-0` is increased. Also, ConDpor using symbolic encoding explores fewer executions than vanilla DPOR in both workloads.

Let us conclude by discussing `TSStack` in more detail, as this data structure was our motivating example in §2. As discussed, this benchmark can be encoded both concretely (using an atomic shared counter), as well as symbolically (using thread-local symbolic counters). As such, we perform the comparison between ConDpor and vanilla DPOR. Based on the `TSStack/100-0` workload in Figure 6, ConDpor explores a single execution. This is because ConDpor encodes timestamping symbolically, and hence push operations are never ordered if there are no pop operations (see § 2.1). Vanilla DPOR, on the other hand, eagerly explores all possible orderings for timestamps, thereby quickly timing out, as depicted in Figure 6. Moreover, as the `TSStack/50-50` workload demonstrates in Figure 6, the symbolic encoding is superior, as ConDpor explores fewer executions than DPOR. This would be even more pronounced using a workload similar to that of client npush+pop in § 2.1.

7 Related Work

As far as handling data nondeterminism is concerned, a lot of research has focused around symbolic execution (e.g., [16, 68, 61]), which uses a symbolic interpreter to examine many program paths at once [43]. As symbolic execution is a static technique and the number of symbolic paths to explore explodes, recent work [31, 19, 32, 21, 53, 66] has focused on dynamic symbolic execution (aka concolic testing), where certain symbolic variables are concretized in order to reduce the state-space size. Despite their success, most symbolic tools do not reason about concurrency and weak memory consistency.

For scheduling nondeterminism, most automated tools are based on either testing or model checking. Testing tools (e.g., [41, 49, 52, 17, 2, 18, 71]) randomly sample the state space of a program in order to detect concurrency bugs, but they do not provide completeness guarantees, nor do they handle data-nondeterminism. Model checkers (e.g., [39, 35]), on the

other hand, do guarantee completeness, as they exhaustively enumerate the state space of a program (up to a bound). In order not to store all the visited program states, model checking is often done "on the fly" (a.k.a. stateless model checking) [55, 30], while also employing partial order reduction [29, 7, 56, 45, 27, 47, 33, 4, 9]. While model checkers can handle data nondeterminism explicitly, explicit enumeration quickly blows up the state space (see §6).

There has been work aiming to handle both scheduling and data nondeterminism (e.g., [28, 65, 70]), either by adding concurrency reasoning to concolic testing, or the other way around. Con2Colic [28] extends CT by introducing interference scenarios to encode the scheduling constraints that lead to new execution paths. Similarly, Guo et al. [34] encode program assertions as symbolic constraints to develop sound methods that prune out redundant executions. Tools like cmbc [24] encode the program into an SAT formula, and offload all symbolic reasoning to an SMT solver. JPF has been extended to incorporate symbolic execution [12]. None of these tools perform optimal partial order reduction for general memory consistency models. Finally, there are several works that attempt to efficiently combine DPOR with symbolically encoded partial-order constraints to reduce redundant exploration (e.g., [11, 42]). Unlike our approach, however, these do not tackle the problem of data nondeterminism within concurrent programs.

Another approach to addressing both scheduling and data non-determinism is maximal path causality (MPC) [69]. MPC uses a coarser equivalence partitioning than ConDpor, as it packs all combinations of schedules and data that reach the same path into a single equivalence class. Upon obtaining a random program interleaving, MPC collects constraints that would lead to the exploration of an unseen path, and explores them by re-running the program.

$$\begin{array}{l} a := \texttt{nondet}() \\ b := \texttt{nondet}() \\ \texttt{if } (a > 0)\ x := 1 \\ \texttt{else} \\ \quad \texttt{if } (b > 0)\ x := 3 \\ \quad \texttt{else } x := 4 \end{array} \quad \Big\| \quad \texttt{assert}(x \neq 2)$$

Apart from not being able to handle weak memory consistency models, MPC is not optimal w.r.t. its partitioning. As an example, consider the program on the right. Assuming MPC first obtains the interleaving $\texttt{C}(a > 0) \cdot \texttt{W}(x, 1) \cdot \texttt{R}(x)$, it will then generate the constraint $a \leq 0 \wedge \texttt{val}(\texttt{R}(x)) = 2$, aiming to uncover the assertion violation. While this constraint (and all paths stemming from it) are infeasible, MPC cannot drop it, as some of the paths under the "else" branch could have been writing $x := 2$.

The closest works to ConDpor are those that aim to combine DPOR and CT. One of the such attempts was jCUTE [60]. Unlike ConDpor, jCUTE uses a non-optimal DPOR (and therefore explores redundant executions), and does not support weak memory consistency. Another work is that of Schemmel et al. [58], who combine quasi-optimal partial order reduction with CT. However, the resulting algorithm (PorSe) is not optimal, does not support weak memory consistency, has exponential memory requirements, and, since it does not use concrete execution, incurs a costly and unnecessary extra query to the SMT solver upon each symbol constraint evaluation. In our extended version [44] (App. A.5), we present a comparison between ConDpor and PorSe using the SV-COMP benchmarks. This comparison shows that, on average, ConDpor reduces the number of explored executions by 72% compared to PorSe across SV-COMP benchmarks.

8 Conclusion

We presented ConDpor, a sound, complete and optimal algorithm that integrates concolic execution into dynamic partial order reduction. ConDpor uses execution graphs for backtracking on both scheduling and symbolic constraints, and offloads all manipulation of symbolic expressions to the underlying language's runtime. ConDpor significantly outperforms DPOR approaches that explicitly enumerate the values of a data domain (even with that domain is small), and can verify challenging concurrent data structures.

References

1 asm / asm · GitLab — gitlab.ow2.org. https://gitlab.ow2.org/asm/asm. [Accessed 29-01-2025].

2 GitHub - openjdk/jcstress. https://github.com/openjdk/jcstress. [Accessed 31-01-2025].

3 GitHub - Z3Prover/z3: The Z3 Theorem Prover — github.com. https://github.com/Z3Prover/z3. [Accessed 30-01-2025].

4 Parosh Abdulla, Mohamed Faouzi Atig, S Krishna, Ashutosh Gupta, and Omkar Tuppe. Optimal stateless model checking for causal consistency. In *TACAS*, pages 105–125. Springer, 2023.

5 Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas. Stateless model checking for TSO and PSO. In *TACAS 2015*, volume 9035 of *LNCS*, pages 353–367, Berlin, Heidelberg, 2015. Springer. doi:10.1007/978-3-662-46681-0_28.

6 Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. Optimal dynamic partial order reduction. In *POPL 2014*, pages 373–384, New York, NY, USA, 2014. ACM. doi:10.1145/2535838.2535845.

7 Parosh Aziz Abdulla, Mohamed Faouzi Atig, Sarbojit Das, Bengt Jonsson, and Konstantinos Sagonas. Parsimonious optimal dynamic partial order reduction. In *CAV 2024*, pages 19–43, Cham, 2024. Springer. doi:10.1007/978-3-031-65630-9_2.

8 Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Tuan Phong Ngo. Optimal stateless model checking under the release-acquire semantics. *Proc. ACM Program. Lang.*, 2(OOPSLA):135:1–135:29, October 2018. doi:10.1145/3276505.

9 Parosh Aziz Abdulla, Ashutosh Gupta, Shankara Narayanan Krishna, and Omkar Tuppe. Dynamic partial order reduction for transactional programs on serializable platforms. In *International Symposium on Automated Technology for Verification and Analysis*, pages 181–202. Springer, 2024. doi:10.1007/978-3-031-78709-6_9.

10 Yehuda Afek, Eli Gafni, and Adam Morrison. Common2 extended to stacks and unbounded concurrency. *Distributed Comput.*, 20(4):239–252, 2007. doi:10.1007/S00446-007-0023-3.

11 Elvira Albert, Maria Garcia de la Banda, Miguel Gómez-Zamalloa, Miguel Isabel, and Peter J. Stuckey. Optimal dynamic partial order reduction with context-sensitive independence and observers. *J. Syst. Softw.*, 202:111730, 2023. doi:10.1016/J.JSS.2023.111730.

12 Saswat Anand, Corina S. Păsăreanu, and Willem Visser. JPF–SE: A symbolic execution extension to java pathfinder. In *TACAS 2007*, pages 134–138, Berlin, Heidelberg, 2007. Springer. doi:10.1007/978-3-540-71209-1_12.

13 Daniel Baier, Dirk Beyer, and Karlheinz Friedberger. Javasmt 3: Interacting with SMT solvers in java. In Alexandra Silva and K. Rustan M. Leino, editors, *CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II*, volume 12760 of *Lecture Notes in Computer Science*, pages 195–208. Springer, 2021. doi:10.1007/978-3-030-81688-9_9.

14 Rudolf Bayer and Mario Schkolnick. Concurrency of operations on b-trees. *Acta Informatica*, 9:1–21, 1977. doi:10.1007/BF00263762.

15 Stefan Bucur, Johannes Kinder, and George Candea. Prototyping symbolic execution engines for interpreted languages. In *ASPLOS 2014*, pages 239–254, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2541940.2541977.

16 Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. Parallel symbolic execution for automated real-world software testing. In *EuroSys 2011*, pages 183–198, New York, NY, USA, 2011. ACM. doi:10.1145/1966445.1966463.

17 Sebastian Burckhardt, Chris Dern, Madanlal Musuvathi, and Roy Tan. Line-Up: A complete and automatic linearizability checker. In Benjamin G. Zorn and Alexander Aiken, editors, *PLDI 2010*, pages 330–340. ACM, 2010. doi:10.1145/1806596.1806634.

18 Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *ASPLOS 2010*, ASPLOS XV, pages 167–178, New York, NY, USA, 2010. ACM. doi:10.1145/1736020.1736040.

19 Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI 2008*, pages 209–224, USA, 2008. USENIX Association. URL: http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf.

20 Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. Exe: Automatically generating inputs of death. *ACM Trans. Inf. Syst. Secur.*, 12(2), December 2008. doi:10.1145/1455518.1455522.

21 Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *IEEE Symposium on Security and Privacy, SP 2012*, pages 380–394. IEEE Computer Society, 2012. doi:10.1109/SP.2012.31.

22 Marek Chalupa, Krishnendu Chatterjee, Andreas Pavlogiannis, Nishant Sinha, and Kapil Vaidya. Data-centric dynamic partial order reduction. *Proc. ACM Program. Lang.*, 2(POPL):31:1–31:30, December 2017. doi:10.1145/3158119.

23 Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. The S2E platform: Design, implementation, and applications. *ACM Trans. Comput. Syst.*, 30(1):2:1–2:49, 2012. doi:10.1145/2110356.2110358.

24 Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *TACAS 2004*, volume 2988 of *LNCS*, pages 168–176, Berlin, Heidelberg, 2004. Springer. doi:10.1007/978-3-540-24730-2_15.

25 Cinzia Di Giusto, Davide Ferré, Laetitia Laversa, and Etienne Lozes. A partial order view of message-passing communication models. *Proc. ACM Program. Lang.*, 7(POPL), January 2023. doi:10.1145/3571248.

26 Mike Dodds, Andreas Haas, and Christoph M. Kirsch. A scalable, correct time-stamped stack. In *POPL 2015*, pages 233–246, New York, NY, USA, 2015. ACM. doi:10.1145/2676726.2676963.

27 Constantin Enea, Dimitra Giannakopoulou, Michalis Kokologiannakis, and Rupak Majumdar. Model checking distributed protocols in must. *Proc. ACM Program. Lang.*, 8(OOPSLA2), October 2024. doi:10.1145/3689778.

28 Azadeh Farzan, Andreas Holzer, Niloofar Razavi, and Helmut Veith. Con2colic testing. In *ESEC/FSE*, pages 37–47. ACM, 2013. doi:10.1145/2491411.2491453.

29 Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *POPL 2005*, pages 110–121, New York, NY, USA, 2005. ACM. doi:10.1145/1040305.1040315.

30 Patrice Godefroid. Model checking for programming languages using VeriSoft. In *POPL 1997*, pages 174–186, New York, NY, USA, 1997. ACM. doi:10.1145/263699.263717.

31 Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *PLDI 2005*, pages 213–223, New York, NY, USA, 2005. ACM. doi:10.1145/1065010.1065036.

32 Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Automated whitebox fuzz testing. In *NDSS*. The Internet Society, 2008. URL: https://www.ndss-symposium.org/ndss2008/automated-whitebox-fuzz-testing/.

33 Henning Günther, Alfons Laarman, Ana Sokolova, and Georg Weissenbacher. Dynamic reductions for model checking concurrent software. In *VMCAI 2017, Paris, France, Proceedings 18*, pages 246–265. Springer, 2017. doi:10.1007/978-3-319-52234-0_14.

34 Shengjian Guo, Markus Kusano, Chao Wang, Zijiang Yang, and Aarti Gupta. Assertion guided symbolic execution of multithreaded programs. In *ESEC/FSE*, pages 854–865. ACM, 2015. doi:10.1145/2786805.2786841.

35 Klaus Havelund and Thomas Pressburger. Model checking JAVA programs using JAVA pathfinder. *Int. J. Soft. Tool. Tech. Transf.*, 2(4):366–381, 2000. doi:10.1007/S100090050043.

36 Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer, and Nir Shavit. A lazy concurrent list-based set algorithm. In *OPODIS 2005*, pages 3–16, Berlin, Heidelberg, 2005. Springer-Verlag. doi:10.1007/11795490_3.

37 Danny Hendler, Nir Shavit, and Lena Yerushalmi. A scalable lock-free stack algorithm. In *SPAA 2004*, pages 206–215, New York, NY, USA, 2004. ACM. doi:10.1145/1007912.1007944.

38 Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming*. Morgan Kaufmann Publishers Inc., 2008.

39 G.J. Holzmann. The model checker spin. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997. doi:10.1109/32.588521.

40 Jeff Huang. Stateless model checking concurrent programs with maximal causality reduction. In *PLDI 2015*, pages 165–174, New York, NY, USA, 2015. ACM. doi:10.1145/2737924.2737975.

41 Karthick Jayaraman, David Harvison, Vijay Ganesh, and Adam Kiezun. jfuzz: A concolic whitebox fuzzer for java. In Ewen Denney, Dimitra Giannakopoulou, and Corina S. Pasareanu, editors, *NFM 2009, Moffett Field, California, USA, April 6-8, 2009*, volume NASA/CP-2009-215407 of *NASA Conference Proceedings*, pages 121–125, 2009.

42 Vineet Kahlon, Chao Wang, and Aarti Gupta. Monotonic partial order reduction: An optimal symbolic partial order reduction technique. In Ahmed Bouajjani and Oded Maler, editors, *CAV 2009, Grenoble, France, Proceedings*, volume 5643 of *Lecture Notes in Computer Science*, pages 398–413. Springer, 2009. doi:10.1007/978-3-642-02658-4_31.

43 James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976. doi:10.1145/360248.360252.

44 Michalis Kokologiannakis, Mohammad Hossein Khoshechin Jorshari, Srinidhi Nagendra, and Rupak Majumdar. Optimal concolic dynamic partial order reduction, June 2025. doi:10.5281/zenodo.15649583.

45 Michalis Kokologiannakis, Iason Marmanis, Vladimir Gladstein, and Viktor Vafeiadis. Truly stateless, optimal dynamic partial order reduction. *Proc. ACM Program. Lang.*, 6(POPL), January 2022. doi:10.1145/3498711.

46 Michalis Kokologiannakis, Iason Marmanis, and Viktor Vafeiadis. Unblocking dynamic partial order reduction. In Constantin Enea and Akash Lal, editors, *CAV 2023*, volume 13964 of *LNCS*, pages 230–250. Springer, 2023. doi:10.1007/978-3-031-37706-8_12.

47 Michalis Kokologiannakis, Iason Marmanis, and Viktor Vafeiadis. SPORE: Combining symmetry and partial order reduction. *Proc. ACM Program. Lang.*, 8(PLDI), June 2024. doi:10.1145/3656449.

48 Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. Model checking for weakly consistent libraries. In *PLDI 2019*, New York, NY, USA, 2019. ACM. doi:10.1145/3314221.3314609.

49 Nikita Koval, Alexander Fedorov, Maria Sokolova, Dmitry Tsitelov, and Dan Alistarh. Lincheck: A practical framework for testing concurrent data structures on JVM. In Constantin Enea and Akash Lal, editors, *CAV 2023*, pages 156–169, Cham, 2023. Springer. doi:10.1007/978-3-031-37706-8_8.

50 Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. Repairing sequential consistency in C/C++11. In *PLDI 2017*, pages 618–632, New York, NY, USA, 2017. ACM. doi:10.1145/3062341.3062352.

51 Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9):690–691, September 1979. doi:10.1109/TC.1979.1675439.

52 Ao Li, Byeongjee Kang, Vasudev Vikram, Isabella Laybourn, Samvid Dharanikota, Shrey Tiwari, and Rohan Padhye. Fray: An efficient general-purpose concurrency testing platform for the jvm, 2025. doi:10.48550/arXiv.2501.12618.

53 Dongge Liu, Gidon Ernst, Toby Murray, and Benjamin I. P. Rubinstein. Legion: Best-first concolic testing (competition contribution). In Heike Wehrheim and Jordi Cabot, editors, *FASE 2020, Dublin, Ireland, Proceedings*, volume 12076 of *Lecture Notes in Computer Science*, pages 545–549. Springer, 2020. doi:10.1007/978-3-030-45234-6_31.

54 Antoni Mazurkiewicz. Trace theory. In *PNAROMC 1987*, volume 255 of *LNCS*, pages 279–324, Berlin, Heidelberg, 1987. Springer. doi:10.1007/3-540-17906-2_30.

55 Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI 2008*, pages 267–280. USENIX Association, 2008. URL: https://www.usenix.org/legacy/events/osdi08/tech/full_papers/musuvathi/musuvathi.pdf.

56 Brian Norris and Brian Demsky. CDSChecker: Checking concurrent data structures written with C/C++ atomics. In *OOPSLA 2013*, pages 131–150. ACM, 2013. doi:10.1145/2509136.2509514.

57 Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-TSO. In *TPHOLs 2009*, pages 391–407. Springer, 2009. doi:10.1007/978-3-642-03359-9_27.

58 Daniel Schemmel, Julian Büning, Cesar Rodriguez, David Laprell, and Klaus Wehrle. Symbolic partial-order execution for testing multi-threaded programs. In Shuvendu K. Lahiri and Chao Wang, editors, *CAV*, volume 12224, pages 376–400. Springer, 2020. doi:10.1007/978-3-030-53288-8_18.

59 Koushik Sen and Gul Agha. CUTE and jCUTE: concolic unit testing and explicit path model-checking tools. In *CAV 2006*, pages 419–423, Berlin, Heidelberg, 2006. Springer-Verlag. doi:10.1007/11817963_38.

60 Koushik Sen and Gul Agha. A race-detection and flipping algorithm for automated testing of multi-threaded programs. In *HVC 2006*, pages 166–182, Berlin, Heidelberg, 2006. Springer-Verlag. doi:10.1007/978-3-540-70889-6_13.

61 Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for c. In *ESEC/FSE 2013*, pages 263–272, New York, NY, USA, 2005. ACM. doi:10.1145/1081706.1081750.

62 Dennis Shasha and Marc Snir. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang. Syst.*, 10(2):282–312, April 1988. doi:10.1145/42190.42277.

63 Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

64 Shuttle. URL: https://github.com/awslabs/shuttle.

65 Stephen F. Siegel, Manchun Zheng, Ziqing Luo, Timothy K. Zirkel, Andre V. Marianiello, John G. Edenhofner, Matthew B. Dwyer, and Michael S. Rogers. Civl: the concurrency intermediate verification language. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2807591.2807635.

66 Zachary Susag, Sumit Lahiri, Justin Hsu, and Subhajit Roy. Symbolic execution for randomized programs. *Proc. ACM Program. Lang.*, 6(OOPSLA2):1583–1612, 2022. doi:10.1145/3563344.

67 SV-COMP. Competition on software verification (SV-COMP), 2019. URL: https://sv-comp.sosy-lab.org/2019/.

68 Nikolai Tillmann and Jonathan de Halleux. Pex–White box test generation for .NET. In *TAP 2008*, pages 134–153, Berlin, Heidelberg, 2008. Springer. doi:10.1007/978-3-540-79124-9_10.

69 Qiuping Yi and Jeff Huang. Concurrency verification with maximal path causality. In *ESEC/FSE 2018*, pages 366–376, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3236024.3236048.

70 Hengbiao Yu, Zhenbang Chen, Xianjin Fu, Ji Wang, Zhendong Su, Jun Sun, Chun Huang, and Wei Dong. Symbolic verification of message passing interface programs. In *ICSE 2020*, pages 1248–1260, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3377811.3380419.

71 Xinhao Yuan, Junfeng Yang, and Ronghui Gu. Partial order aware concurrency sampling. In Hana Chockler and Georg Weissenbacher, editors, *CAV 2018*, pages 317–335, Cham, 2018. Springer International Publishing. doi:10.1007/978-3-319-96142-2_20.

Coverage Games

Orna Kupferman
School of Computer Science and Engineering, The Hebrew University, Jerusalem, Israel
Noam Shenwald
School of Computer Science and Engineering, The Hebrew University, Jerusalem, Israel

Abstract

We introduce and study *coverage games* – a novel framework for *multi-agent planning* in settings in which a system operates several agents but do not have full control on them, or interacts with an environment that consists of several agents.

The game is played between a *coverer*, who has a set of objectives, and a *disruptor*. The coverer operates several agents that interact with the adversarial disruptor. The coverer wins if every objective is satisfied by at least one agent. Otherwise, the disruptor wins.

Coverage games thus extend traditional two-player games with multiple objectives by allowing a (possibly dynamic) decomposition of the objectives among the different agents. They have many applications, both in settings where the system is the coverer (e.g., multi-robot surveillance, coverage in multi-threaded systems) and settings where it is the disruptor (e.g., prevention of resource exhaustion, ensuring non-congestion). We study the theoretical properties of coverage games, including determinacy, and the ability to a priori decompose the objectives among the agents. We solve the problems of deciding whether the coverer or the disruptor wins, analyze their tight complexity, and consider useful special cases.

2012 ACM Subject Classification Theory of computation → Formal languages and automata theory; Computing methodologies → Multi-agent planning

Keywords and phrases Two-Player Games, ω-Regular Objectives, Coverage, Planning

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2025.27

Funding *Orna Kupferman*: This research is supported by the European Research Council, Advanced Grant ADVANSYNT.

1 Introduction

Synthesis is the automated construction of a system from its specification [23]. A *reactive* system interacts with its environment and has to satisfy its specification in all environments [15]. A useful way to approach synthesis of reactive systems is to consider the situation as a *two-player game* between the system and the environment [3]. The game is played on a graph whose vertices are partitioned between the players. Starting from an initial vertex, the players jointly move a token and generate a *play*, namely a path in the graph, with each player deciding the successor vertex when the token reaches a vertex she owns. The system wins if it has a strategy to ensure that no matter how the environment moves the token, the generated play satisfies an objective induced by the specification.

Reactive systems typically have to satisfy a conjunction of requirements, giving rise to games with *multiple objectives*. For example, in games with *generalized Büchi* objectives [6], the specification to the system induces a set $\{\alpha_1, \ldots, \alpha_m\}$ of subsets of vertices, and a play satisfies the specification if it visits infinitely often each of the sets α_l, for $1 \leq l \leq m$. Researchers have also considered *generalized parity* [7], *generalized reachability* [13], and *generalized reactivity* (GR(1)) [22, 6] objectives. In addition to the deterministic turn-based setting, games with multiple objectives have been studied in various richer settings, like concurrent, stochastic, and energy games [4, 8, 5, 29], and also allow *weighted* multiple objectives [19].

© Orna Kupferman and Noam Shenwald;
licensed under Creative Commons License CC-BY 4.0

The need to satisfy multiple objectives is also at the heart of *planning* in multi-agent systems [25]. In a typical setting there, the agents constitute a fleet of robots, drones, or autonomous cars that needs to accomplish various tasks, such as exploring an unknown terrain or accommodating requests from users at different locations [10, 25, 9, 24, 26]. For example, in *adversarial patrolling* [20], a fleet of robots needs to detect penetrations into a guarded area; in the *offline-coverage* problem [1], the robots are required to visit every point in a work area, typically within the shortest possible time [8, 25, 9]; and in *pursuit evasion* [17], a robot aims to perform some task while avoiding a collision with an adversarial robot.

While the problems studied in planning in multi-agent systems involve adversarial factors such as physical limitations of the agents, bounded resources, or possible collisions, they assume full control over the agents. That is, they do not include an adversarial behavior of the agents themselves in the form studied in reactive synthesis.

We introduce and study *coverage games* – a framework for reasoning about planning tasks in which the system does not have full control over the agents. Formally, a coverage game is a two-player game with a set $\beta = \{\alpha_1, \ldots, \alpha_m\}$ of objectives, in which one player – the covering player (named *Coverer*) operates a number k of agents that together aim to cover all the objectives in β. In the beginning of the game, k tokens are placed on the initial vertex. Each agent is responsible for one token, and as in standard two-player games, the interaction of each agent with the second player – the disrupting player (named *Disruptor*) generates a play in the graph. The *outcome* of the game is then a set of k plays – one for each agent. The goal of Coverer is for these plays to cover all the objectives in β. That is, for all $1 \leq l \leq m$, there is an agent $1 \leq i \leq k$ such that α_l is satisfied in the play generated by the interaction of Agent i with Disruptor.

A *covering strategy* for Coverer is a vector of strategies, one for each agent, that ensures that no matter how Disruptor behaves, each objective is satisfied in at least one play in the outcome. In the *coverage problem*, we are given a game graph G, a set β of objectives, and a number $k \geq 1$ of agents, and we have to decide whether Coverer has a covering strategy.

Note that when $k = 1$, a coverage game coincides with a standard two-player game with multiple objectives. Also, it is not hard to see that when $k \geq m$, namely there are more agents than objectives, then a dominant strategy (in the game-theoretic sense) for Coverer allocates each objective to a different agent. The interesting cases are when $1 < k < m$, in which case the objectives need to be partitioned, possibly dynamically, among the agents.

Coverage games significantly extend the settings addressed in current studies of planning. For example, with Büchi underlying objectives, one can reason about *multi-robot surveillance*, where we need to ensure that each set of critical locations is visited infinitely often in at least one robot's patrol route. There, vertices owned by Disruptor correspond to locations in which we cannot control the movement of the robots, as well as positions in which the robots may encounter obstacles or experience a change in the landscape. Then, in *cyber-security systems*, the agents represent defense mechanisms, and the goal is to ensure that each set of potential attack vectors is mitigated infinitely often by at least one defense, countering an adversarial hacker. As another example, in *multi-threaded systems*, the agents are the processes, and we may want to ensure that each set of critical resources is accessed infinitely often, in all environments. Likewise, in *testing*, we want to activate and cover a set of functionalities of a software in all input sequences [11].

Unlike traditional two-player games, where the environment can be viewed as a system that aims to realize the negation of the specification, in coverage games Disruptor's objective is not to cover the negation of the objectives in β, but rather to prevent Coverer from covering all the objectives in β. Formally, a *disrupting strategy* for Disruptor is a strategy such that

for all strategies of Coverer, there is at least one objective in β that is not satisfied in each of the k generated plays. Note that while Coverer activates several agents, Disruptor follows a single strategy. This corresponds to environments that model the physical conditions of a landscape or servers whose response to users depends only on the interaction and not on the identity of the user. Then, in the *disruption problem*, we are given G, β, and k, and we have to decide whether Disruptor has a disrupting strategy. While two-player games are determined, in the sense that in all games, one of the two players has a winning strategy, we are going to see that coverage games need not be determined.

Viewing Disruptor as the system and the agents as operated by the environment further motivates the study of coverage games. For example, in *intrusion detection* systems, the agents of the environment are potential intruders, and the system must ensure that at least one set of critical access points remains blocked. In *cloud computing*, the environment consists of client processes competing for resources, and the system's goal is to prevent resource exhaustion. Finally, in *traffic management*, the environment agents are the vehicles, and the system ensures that at least one route remains uncongested. Note that now, Disruptor following a single strategy corresponds to systems like traffic controllers or vending machines, where the same policy is applied to all cars or customers that exhibit identical behavior.

A model related to coverage games from the point of view of the disruptor is *population games* [2]. These games focus on controlling a homogeneous population of agents that have the same non-deterministic behavior (see also [12, 14]). The edges of a two-player population game graph are labeled by actions. The game is nondeterministic, thus different edges that leave the same vertex may be labeled by the same action. As in the disruption problem, each agent moves a token along the graph. In each turn, a controller chooses the same action for all agents, and each agent decides how to resolve nondeterminism and proceeds to a successor vertex along an edge labeled with the action. The goal of the controller is to synchronize all the agents to reach a target state. The technical details of population games are very different from those of coverage games. Indeed, there, the controller takes the same action in all of its interactions with the agents, and the type of objectives are different from the coverage and disruption objectives in coverage games.

We study the coverage and disruption problems in coverage games with *Büchi* and *co-Büchi* objectives. We first study the theoretical aspects of coverage games. We warm up with games in which the number of agents enables an easy reduction to usual two-player games, and continue to *one-player coverage games*, where all vertices are owned by one player. In particular, when all vertices are owned by Coverer, the coverage problem boils down to finding k paths that cover all the objectives in β, and our complexity results are aligned with the NP-hardness of different variants of the *coverage path planning* problem for robot swarms [1].

We then continue to the general setting, show that it is undetermined, and examine the problem of an a-priori decomposition of the objectives in β among the agents. We show that the objectives cannot be allocated in advance, and characterize covering strategies for Coverer as ones in which all agents satisfy all objectives, as long as such decomposition is impossible. The characterization is the key to our upper bounds for the complexity of the coverage problem. We prove that the coverage problem is PSPACE-complete (Theorems 10 and 11), and that the disruption problem is Σ_2^{P}-complete (Theorems 15, 20, and 21), for both Büchi and co-Büchi objectives.

From a technical point of view, for the upper bounds, the main challenging results are the characterization of the decomposability of objectives and the recursive algorithm it entails (Theorems 9 and 10), as well as the ability to work with a short symbolic representation of strategies that are not memoryless (Lemma 19). For the lower bounds, the main technical issue is utilizing the agents to model satisfying assignments in Boolean satisfaction problems.

In Section 7 we discuss possible variants and extensions of coverage games. Beyond classical extensions of the underlying two-player games (e.g., concurrency, stochastic settings, partial visibility, etc.) and to the objectives (e.g., richer winning conditions, weighted objectives, etc.), as well as classical extensions from planning (optimality of agents and their resources), we focus on elements that have to do with the operation of several agents: communication among the agents (in our setting, the strategies of the agents are independent of each other), and the ability of Disruptor to also use different agents.

2 Preliminaries

2.1 Two-player games

A *two-player game graph* is a tuple $G = \langle V_1, V_2, v_0, E\rangle$, where V_1, V_2 are disjoint sets of vertices, owned by Player 1 and Player 2, respectively, and we let $V = V_1 \cup V_2$. Then, $v_0 \in V$ is an initial vertex and $E \subseteq V \times V$ is a total edge relation, thus for every $v \in V$, there is $u \in V$ such that $\langle v, u\rangle \in E$. The size of G, denoted $|G|$, is $|E|$, namely the number of edges in it. When we draw game graphs, the vertices in V_1 and V_2 are drawn as circles and squares, respectively.

In a beginning of a play in the game, a token is placed on v_0. Then, in each turn, the player that owns the vertex that hosts the token chooses a successor vertex and moves the token to it. Together, the players generate a *play* $\rho = v_0, v_1, \ldots$ in G, namely an infinite path that starts in v_0 and respects E: for all $i \geq 0$, we have that $\langle v_i, v_{i+1}\rangle \in E$.

A *strategy* for Player i is a function $f_i : V^* \cdot V_i \to V$ that maps prefixes of plays that end in a vertex owned by Player i to possible extensions of the play. That is, for every $\rho \in V^*$ and $v \in V_i$, we have that $\langle v, f_i(\rho \cdot v)\rangle \in E$. Intuitively, a strategy for Player i directs her how to move the token, and the direction may depend on the history of the game so far. A strategy is *memoryless* if its choices depend only on the current vertex and are independent of the history of the play. Accordingly, we describe a memoryless strategy for Player i by a function $f_i : V_i \to V$.

A *profile* is a tuple $\pi = \langle f_1, f_2\rangle$ of strategies, one for each player. The *outcome* of a profile $\pi = \langle f_1, f_2\rangle$ is the play obtained when the players follow their strategies in π. Formally, $\mathsf{outcome}(\pi) = v_0, v_1, \ldots \in V^\omega$ is such that for all $j \geq 0$, we have that $v_{j+1} = f_i(v_0, v_1, \ldots, v_j)$, where $i \in \{1, 2\}$ is such that $v_j \in V_i$.

A *two-player game* is a pair $\mathcal{G} = \langle G, \psi\rangle$, where $G = \langle V_1, V_2, v_0, E\rangle$ is a two-player game graph, and ψ is an objective for Player 1, specifying a subset of V^ω, namely the set of plays in which Player 1 wins. We discuss different types of objectives below. The game is zero-sum, thus Player 2 wins when the play does not satisfy ψ. A strategy f_1 is a *winning strategy* for Player 1 if for every strategy f_2 for Player 2, we have that Player 1 wins in $\langle f_1, f_2\rangle$, thus $\mathsf{outcome}(\langle f_1, f_2\rangle)$ satisfies ψ. Dually, a strategy f_2 for Player 2 is a winning strategy for Player 2 if for every strategy f_1 for Player 1, we have that Player 2 wins in $\langle f_1, f_2\rangle$. We say that Player i *wins* in $\mathcal{G}$ if she has a winning strategy. A game is *determined* if Player 1 or Player 2 wins it.

For a play $\rho = v_0, v_1, \ldots$, we denote by $\mathit{inf}(\rho)$ the set of vertices that are visited infinitely often along ρ. That is, $\mathit{inf}(\rho) = \{v \in V : \text{there are infinitely many } i \geq 0 \text{ such that } v_i = v\}$. For a set of vertices $\alpha \subseteq V$, a play ρ satisfies the *Büchi* objective α iff $\mathit{inf}(\rho) \cap \alpha \neq \emptyset$. The objective dual to Büchi is *co-Büchi*. Formally, a play ρ satisfies a co-Büchi objective α iff $\mathit{inf}(\rho) \cap \alpha = \emptyset$. We use $\gamma \in \{\mathrm{B}, \mathrm{C}\}$ to denote the different objective types.

For a play ρ and a set of objectives $\beta = \{\alpha_1, \ldots, \alpha_m\}$, we denote by $\mathsf{sat}(\rho, \beta)$ the set of objectives $\alpha_i \in \beta$ that are satisfied in ρ. An *All objective* is a set β of objectives, all of the same type. A play ρ satisfies an All objective β iff ρ satisfies every objective in β. That is,

if $\mathsf{sat}(\rho, \beta) = \beta$. The objective dual to All is *Exists*.[1] Formally, a play ρ satisfies an Exists objective β iff ρ satisfies at least one objective in β. That is, if $\mathsf{sat}(\rho, \beta) \cap \beta \neq \emptyset$. Note that an All-co-Büchi objective β is equal to the co-Büchi objective $\cup\beta$.

2.2 Two-player multi-agent coverage games

In coverage games, Player 1 operates a number of agents. All agents play against Player 2, and so the outcome of the game is a set of plays – one for each agent. Player 1 has multiple objectives, and her goal is to cover all the objectives, in the sense that each objective is satisfied in at least one of the plays in the outcome. Accordingly, we refer to Player 1 as the covering player, named *Coverer*, and refer to Player 2 as the disrupting player, named *Disruptor*.

Formally, for $\gamma \in \{\text{B,C}\}$, a *two-player multi-agent γ-coverage game* (γ-CG, for short) is a tuple $\mathcal{G} = \langle G, k, \beta \rangle$, where G is a two-player game graph, $k \in \mathbb{N}$ is the number of agents Coverer operates, and $\beta = \{\alpha_1, \ldots, \alpha_m\}$ is a set of objectives of type γ. When γ is not important or is clear from the context, we omit it and refer to $\mathcal{G}$ as a CG.

For $k \geq 1$, let $[k] = \{1, \ldots, k\}$. A *strategy* for Coverer is a tuple $F_1 = \langle f_1^1, \ldots, f_1^k \rangle$ of k strategies for Coverer in the game graph G. We assume that Disruptor uses the same strategy f_2 against all the agents of Coverer. A *profile* is a tuple $\pi = \langle F_1, f_2 \rangle$ of strategies $F_1 = \langle f_1^1, \ldots, f_1^k \rangle$ for Coverer and f_2 for Disruptor. The outcome of π is the tuple of the k plays generated when Coverer's agents and Disruptor follow their strategies. Formally, $\mathsf{outcome}(\pi) = \langle \rho_1, \ldots, \rho_k \rangle$, where $\rho_i = \mathsf{outcome}(\langle f_1^i, f_2 \rangle)$, for every $i \in [k]$.

We say that a profile π *covers* β iff every objective in β is satisfied in at least one play in $\mathsf{outcome}(\pi)$. That is, if $\cup\{\mathsf{sat}(\rho, \beta) : \rho \in \mathsf{outcome}(\pi)\} = \beta$. Equivalently, for every $j \in [m]$, there exists $i \in [k]$ such that α_j is satisfied in $\mathsf{outcome}(\langle f_1^i, f_2 \rangle)$. A strategy F_1 for Coverer is a *covering* strategy in $\mathcal{G}$ iff for every strategy f_2 for Disruptor, the profile $\langle F_1, f_2 \rangle$ covers β. Then, we say that Coverer wins $\mathcal{G}$ iff she has a covering strategy in $\mathcal{G}$. A strategy f_2 for Disruptor is a *disrupting* strategy in $\mathcal{G}$ iff for every strategy F_1 for Coverer, the profile $\langle F_1, f_2 \rangle$ does not cover β. Then, we say that Disruptor wins $\mathcal{G}$ iff she has a disrupting strategy in $\mathcal{G}$. Finally, we say that a γ-CG $\mathcal{G}$ is *determined* if Coverer has a covering strategy in $\mathcal{G}$ or Disruptor has a disrupting strategy in $\mathcal{G}$.

In the *coverage problem for CGs*, we have to decide whether Coverer has a covering strategy in a given CG. In the *disruption problem for CGs*, we have to decide whether Disruptor has a disrupting strategy in a given CG.

▶ **Example 1.** Consider the Büchi CG $\mathcal{G} = \langle G, 2, \beta \rangle$, with $\beta = \{\alpha_1, \alpha_2, \alpha_3\}$, for the game graph G appearing in Figure 1, and $\alpha_1 = \{u_1, u_2, u_3\}$ (red), $\alpha_2 = \{m_1, u_2, d_3\}$ (green), and $\alpha_3 = \{d_1, d_2, d_3\}$ (yellow). Thus, Coverer should direct two agents in a way that ensures that the three colors are visited infinitely often.

Coverer has a covering strategy in $\mathcal{G}$: In v_0, the tokens of both agents move together, visiting the three colors in a round-robin fashion. If the play stays forever in the left sub-game, β is covered. If Disruptor decides to leave the left sub-game and directs the tokens to v_1 or v_2, the agents split β between them. One agent covers two colors (green and red in u_2, in case the tokens are in v_1, or green and yellow in d_3, in case the tokens are in v_2), and the second agent covers one color (yellow or red, respectively). Thus, β is covered in this case too. ◀

[1] All-Büchi objectives are traditionally referred to as *generalized Büchi*. Then, their dual Exists co-Büchi objectives are referred to as *generalized co-Büchi* [18]. We opt to follow a terminology with All and Exists, making the quantification within the satisfaction requirement clearer.

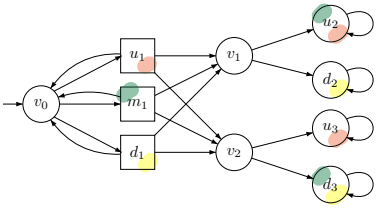


Figure 1 The Büchi CG $\mathcal{G}$.

3 Special Cases of Coverage Games

As a starter, in this section we examine two special cases of CGs: game graphs in which there is only one player, and games in which the number of agents enables an easy reduction to traditional two-player games.

3.1 One-player CGs

We start with one-player CGs, thus when one of the players owns all the vertices. Formally, $\mathcal{G} = \langle G, k, \beta \rangle$, with $G = \langle V_1, V_2, v_0, E \rangle$ is such that $V_2 = \emptyset$ (equivalently, $V = V_1$) or $V_1 = \emptyset$ (equivalently, $V = V_2$). Note that one-player CGs are determined, in the sense that whenever Coverer does not have a covering strategy, Disruptor has an (empty) disrupting strategy, and vice versa.

Consider first the case Coverer has full control. Thus, $G = \langle V, \emptyset, v_0, E \rangle$. It is easy to see that there, covering β amounts to finding k paths in G such that each objective $\alpha_i \in \beta$ is satisfied in at least one path. A path in G is *lasso-shaped* if it is of the form $p \cdot q^\omega$, with $p \in V^*$ and $q \in V^+$. The length of $p \cdot q^\omega$ is defined as $|p| + |q|$.

▶ **Theorem 2.** *The coverage problem for Büchi or co-Büchi CGs with $V = V_1$ is NP-complete.*

Proof. We start with the upper bounds. Consider a CG $\mathcal{G} = \langle G, k, \beta \rangle$ with $V = V_1$. An NP algorithm guesses k lasso-shaped paths in G of length of at most $|V| \cdot |\beta|$, and checks that every objective $\alpha_i \in \beta$ is satisfied in at least one of them. Clearly, if such k paths exist, then Coverer has a covering strategy. Also, checking lasso-shaped paths of length $|V| \cdot |\beta|$ is sufficient. Indeed, for both $\gamma \in \{\text{B,C}\}$, if there exists a path in G that satisfies a subset $\beta' \subseteq \beta$ of γ objectives, then there also exists a cycle in G of length of at most $|V| \cdot |\beta'|$ that visits or avoids all the sets in β'. Since $|\beta'| \leq |\beta|$, the bound follows.

For the lower bound, we describe (easy) reductions from the vertex-cover problem. Consider an undirected graph $G = \langle V, E \rangle$, and $k \leq |V|$. Recall that a vertex cover for G is a set $U \subseteq V$ such that for all edges $\{u, v\} \in E$, we have that $\{u, v\} \cap U \neq \emptyset$. For both $\gamma \in \{\text{B,C}\}$, we construct a γ-CG $\mathcal{G} = \langle G', k, \beta \rangle$ with $V = V_1$ such that Coverer has a covering strategy in $\mathcal{G}$ iff there exists a vertex cover of size at most k in G.

The game graph G' is independent of E and consists of the vertices in V and a new initial vertex v_0. The only edges are self-loops in the vertices in V, and edges from v_0 to all vertices in V. Accordingly, the k agents of Coverer essentially choose k vertices in V. The objectives in β then require these k vertices to be a vertex cover. For $\gamma = \text{B}$, we define $\beta = E$. That is, for every edge $e = \{u, v\} \in E$, the objective e is to visit one of the vertices that correspond to u or v infinitely often. Accordingly, a covering strategy for Coverer induces a vertex cover, and vice versa. For $\gamma = \text{C}$, we define $\beta = \{V \setminus e : e \in E\}$. Since each play ρ is eventually trapped in a self-loop in a vertex in V, the obtained game coincides with the one for $\gamma = \text{B}$.

A detailed description of the reduction and its proof can be found in Appendix A.1. ◀

We continue to the case all the vertices in the game are owned by Disruptor. It is easy to see that there, the fact Coverer has several agents plays no role, as all tokens are going to traverse the same play, chosen by Disruptor. Consequently, searching for a disrupting strategy for Disruptor is easy.

▶ **Theorem 3.** *The disruption problem for Büchi or co-Büchi CGs with $V = V_2$ is NLOGSPACE-complete.*

Proof. Consider a CG $\mathcal{G} = \langle G, k, \beta \rangle$ with $G = \langle \emptyset, V, v_0, E \rangle$. Since Disruptor directs all the tokens together, she has a disrupting strategy iff there is a path in G that satisfies the Exists-$\tilde{\gamma}$ objective dual to β, namely a path that does not satisfy one of the objectives in β. We argue that the latter can be done in NLOGSPACE, for all $\gamma \in \{\text{B,C}\}$. Indeed, an NLOGSPACE algorithm can guess a set $\alpha \in \beta$ and then search for a lasso-shaped path that does not satisfy α. In case $\gamma = \text{B}$, the vertices of the lasso loop should be disjoint from α. In case $\gamma = \text{C}$, the algorithm guesses a vertex $v \in \alpha$ and checks that v appears in the lasso loop. In both cases, it is possible to search for paths of length at most $|V|$. This can be done in NLOGSPACE by guessing the vertices along the path one by one.

Hardness in NLOGSPACE follows from an easy reduction from the non-emptiness problem for nondeterministic ExistsC and ExistsB word automata [28]. ◀

3.2 CGs with one or many agents

Consider a CG $\mathcal{G} = \langle G, k, \beta \rangle$. When $k = 1$, the CG $\mathcal{G}$ is equivalent to the two-player game with the All objective β. Indeed, the single play induced by the strategy of the single agent has to cover all the objectives in β. Thus, the case $k = 1$ can be solved in PTIME, serving as a lower bound for the general case, or more precisely, as a reference point to the study of the complexity for the general case.

When $k \geq |\beta|$, there is no need to assign more than one objective in β to each of the agents. Formally, we have the following.

▶ **Lemma 4.** *Consider a γ-CG $\mathcal{G} = \langle G, k, \beta \rangle$ with $\gamma \in \{B,C\}$ and $k \geq |\beta|$, and the following statements.*

(C1) *Coverer has a covering strategy in $\mathcal{G}$.*
(C2) *For every $\alpha_i \in \beta$, Player 1 wins the two-player γ-game $\mathcal{G}_i = \langle G, \alpha_i \rangle$.*
(C3) *There is $\alpha_i \in \beta$ such that Player 2 wins the two-player γ-game $\mathcal{G}_i = \langle G, \alpha_i \rangle$.*
(C4) *Disruptor has a disrupting strategy in $\mathcal{G}$.*
Then, $(C1)$ iff $(C2)$, $(C3)$ iff $(C4)$, and $(C1)$ and $(C4)$ complement each other.

Proof. First note that since two-player γ-games are determined, then $(C2)$ and $(C3)$ complement each other. Also, as it is impossible for both Coverer to have covering strategy and Disruptor to have a disrupting strategy, $(C1)$ and $(C4)$ contradicts each other. Hence, it is sufficient to prove that $(C2)$ implies $(C1)$ and that $(C3)$ implies $(C4)$.

Assume that $(C2)$ holds. For every $\alpha_i \in \beta$, let f_1^i be a winning strategy for Coverer in $\mathcal{G}_i = \langle G, \alpha_i \rangle$. It is easy to see that $F_1 = \langle f_1^1, \ldots, f_1^{|\beta|}, f_1^{|\beta|+1}, \ldots, f_1^k \rangle$ is a covering strategy for Coverer, for arbitrary strategies $f_1^{|\beta|+1}, \ldots, f_1^k$ for agents $|\beta|+1, \ldots, k$. Thus, Coverer has a covering strategy, and $(C1)$ holds.

Assume now that $(C3)$ holds. Thus, there is $\alpha_i \in \beta$ such that Disruptor wins the two-player γ-game $\mathcal{G}_i = \langle G, \alpha_i \rangle$. Let f_2 be the winning strategy for Disruptor in $\mathcal{G}_i$. Then, it must be that $\alpha_i \notin \mathsf{sat}(\rho, \beta)$ for every strategy F_1 for Coverer and every play $\rho \in \mathsf{outcome}(\langle F_1, f_2 \rangle)$. Thus, Disruptor has a disrupting strategy and $(C4)$ holds. ◀

By Lemma 4, deciding CGs with $k \geq |\beta|$ agents can be reduced to deciding $|\beta|$ two-player games, thus such CGs are determined and can be decided in PTIME [21, 27, 30].

When $1 < k < |\beta|$, Coverer has to partition the objectives in β among the different agents. As we study in Section 4, this makes the setting much more challenging and interesting.

4 Properties of Coverage Games

In this section we study theoretical properties of CGs. We start with determinacy and show that CGs need not be determined. We then define and study the *decomposability* of objectives in CGs, namely the ability to a-priori partition the objectives among the agents. We argue that decomposability plays a key role in reasoning about CGs.

4.1 CGs need not be determined

Recall that a CG is determined if Coverer has a covering strategy or Disruptor has a disrupting strategy. We show that unlike two-player games, CGs need not be determined. Moreover, undeterminacy holds already for a CG with only two agents and three objectives. Note that this is tight, as CGs with a single agent or only two objectives belong to the $k = 1$ or $k \geq |\beta|$ cases studied in Section 3.2, and are thus determined. Intuitively, the undeterminacy of CGs follows from the incomplete information of Coverer's agents about all the plays in the outcome, as well as the inability of Disruptor to make use of this incomplete information. Formally, we have the following.

▶ **Theorem 5.** *For all $\gamma \in \{B,C\}$, we have that γ-CGs need not be determined. Moreover, undeterminacy holds already for a γ-CG with only two agents and three objectives.*

Proof. For all $\gamma \in \{\text{B,C}\}$, we describe a γ-CG $\mathcal{G} = \langle G, 2, \{\alpha_1, \alpha_2, \alpha_3\}\rangle$ such that neither Coverer has a covering strategy, nor Disruptor has a disrupting strategy. The game graph G appears in Figure 2.

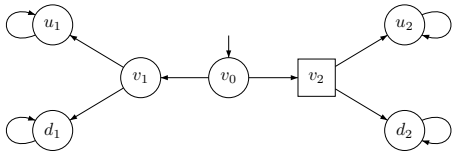


Figure 2 An undetermined coverage game.

When $\gamma = B$, we define $\alpha_1 = \{u_1, u_2\}$, $\alpha_2 = \{d_1, d_2\}$, and $\alpha_3 = \{u_2, d_2\}$. Intuitively (see detailed proof in Appendix A.2), Coverer has no covering strategy as α_3 requires one agent to move to v_2, and no matter how Coverer directs the other agent, Disruptor can move the tokens that reach v_2 so that only one of α_1 and α_2 is satisfied. Also, Disruptor does not have a disrupting strategy, as no matter how the strategy moves token that reach v_2, Coverer can direct the agents so that all objectives are covered.

When $\gamma = C$, we define $\alpha_1 = \{u_1, u_2\}$, $\alpha_2 = \{d_1, d_2\}$, and $\alpha_3 = \{u_1, d_1\}$. Since the only edges from the vertices u_1, d_1, u_2, and d_2 are self-loops, the obtained co-Büchi game coincides with the Büchi game analyzed above, and we are done. ◀

4.2 Decomposability of objectives

A key challenge for Coverer in winning a CG $\mathcal{G} = \langle G, k, \beta\rangle$ with $2 \leq k < |\beta|$ is the need to partition the objectives in β among her k agents. In this section we show that in general, Coverer cannot *a-priori* partition β among the k agents. Moreover, decompositions of β are related to decompositions of G, which are the key to our algorithms for reasoning about CGs.

Consider a game graph $G = \langle V_1, V_2, v_0, E\rangle$. For a vertex $v \in V_1$, let $G^v = \langle V_1, V_2, v, E\rangle$ be G with initial vertex v. Consider a CG $\mathcal{G} = \langle G, k, \beta\rangle$, with $\beta = \{\alpha_1, \ldots, \alpha_m\}$. Let $A = [k]$. For $1 \leq l \leq k$ and a vertex $v \in V_1$, we say that β is (k,l)-*decomposable* in v if Coverer can decompose the task of covering β in G^v by the agents in A to l sub-tasks, each assigned to a different subset of A. Formally, β is (k,l)-decomposable in v if there exists a partition $\beta_1, \ldots, \beta_l$ of β and a partition $A_1, \ldots, A_l$ of A to nonempty sets, such that for every $i \in [l]$, Coverer wins the CG $\mathcal{G}_i^v = \langle G^v, |A_i|, \beta_i\rangle$, namely the game that starts in v and in which the agents in A_i have to cover the objectives in β_i. When $v = v_0$, we say that β is (k,l)-*decomposable* in G.

We start with some easy observations. First, note that, by definition, β is $(k,1)$-*decomposable* in G iff Coverer wins $\mathcal{G}$. Indeed, when $l = 1$, we have that $A_1 = A$ and $\beta_1 = \beta$, thus we do not commit on a decomposition of β, and the definitions coincide. Then, β is (k,k)-*decomposable* in G iff there exists a partition of β to k sets $\beta_1, \ldots, \beta_k$ such that for every $1 \leq i \leq k$, Coverer has a strategy that satisfies the All objective β_i. Indeed, when $l = k$, each of the sets A_i is a singleton, thus the single agent in A_i has to satisfy all the objectives in β_i. Finally, the bigger l is, the more refined is the partition of β to which Coverer commits, making her task harder. Formally, we have the following (see Appendix A.3 for the proof and Example 1 for a CG with no a-priori decomposition):

▶ **Theorem 6.** *For all $\gamma \in \{B, C\}$, we have the following.*

- *Consider a γ-CG $\mathcal{G} = \langle G, k, \beta\rangle$. For every $l \in [k]$ and vertex v of G, if β is (k,l)-decomposable in v, then β is (k,l')-decomposable in v, for all $l' \leq l$.*
- *For every $k \geq 2$, there exists a γ-CG $\mathcal{G} = \langle G, k, \beta\rangle$ with $|\beta| = k+1$ such that Coverer wins $\mathcal{G}$ but β is not (k,l)-decomposable in G, for all $l > 1$.*
- *For every $k \geq 2$ and $1 \leq l < k$, there exists a γ-CG $\mathcal{G} = \langle G, k, \beta\rangle$ such that β is (k,l)-decomposable in G but is not $(k,l+1)$-decomposable in G.*

Consider a vertex $v \in V_1$. For $l \in [k]$, we say that v is a (k,l)-*fork* for β if Coverer has a covering strategy $F_1 = \langle f_1^1, \ldots, f_1^k\rangle$ in G^v such that different agents are sent to l different successors of v. Thus, there exist l successors $v_1, \ldots, v_l$ of v, and for all $i \in [l]$, there is at least one agent $j \in [k]$ such that $f_1^j(v) = v_i$. Note that we do not require f_1^i to be memoryless and refer here to the strategy in the first round in G^v. The following lemma then follows from the definitions.

▶ **Lemma 7.** *Consider a CG $\mathcal{G} = \langle G, k, \beta\rangle$. For every vertex $v \in V_1$ of G and $l \in [k]$, if v is a (k,l)-fork for β, then β is (k,l)-decomposable in v.*

Proof. Consider a vertex $v \in V_1$ and $1 \leq l \leq k$ such that v is a (k,l)-fork for β. Let $F_1 = \langle f_1^1, \ldots, f_1^k\rangle$ be a winning strategy for Coverer that sends the agents to l different successors $v_1, \ldots, v_l$ of v. For every $1 \leq i \leq l$, let A_i be the set of agents $j \in [k]$ such that $f_1^j(v) = v_i$, and let $F_1^i = \{f_1^j\}_{j \in A_i}$ be the set of strategies of the agents sent to v_i. Let β_i be the set objectives of β that F_1^i covers. That is, $\alpha_j \in \beta_i$ iff for all strategies f_2 of Disruptor, there is a play in $\mathsf{outcome}(F_1^i, f_2)$ that satisfies α_j. Equivalently, $\beta_i \subseteq \beta$ is the maximal subset of β such that F_1^i is a winning strategy for Coverer in $\langle G_{v_i}, |A_i|, \beta_i\rangle$. It is then sufficient to show that $\bigcup_{i \in [l]} \beta_i = \beta$. Assume by contradiction that there exists an objective $\alpha_j \in \beta$ such that $\alpha_j \notin \beta_i$, for every $i \in [l]$, which implies there exists a strategy f_2^i for Disruptor in G_{v_i} such that α_j is not covered in $\mathsf{outcome}(\langle F_1^i, f_2^i\rangle)$. Let f_2 be the strategy for Disruptor from v that follows f_2^i from v_i, for every $i \in [l]$. It is easy to see that α_j is not covered in $\mathsf{outcome}(\langle F_1, f_2\rangle)$, contradicting the fact F_1 is a winning strategy. ◀

We say that a vertex $v \in V_1$ is a *fork* if it is a (k,l)-fork for β, for some $2 \leq l \leq k$. We denote by $F \subseteq V_1$ the set of vertices that are forks. Let $V_{avoid} \subseteq V$ be the set of vertices from which Disruptor can avoid reaching forks. That is, $v \in V_{avoid}$ if there exists a strategy f_2 for Player 2 from v such that for every strategy f_1 for Player 1, the play $\mathsf{outcome}(\langle f_1, f_2\rangle)$ does not reach F. Finally, let $G_{avoid} = \langle V_1', V_2', v_0, E'\rangle$ be the sub-graph of G with vertices in V_{avoid}. That is, $V_1' = V_1 \cap V_{avoid}$, $V_2' = V_2 \cap V_{avoid}$, and $E' = E \cap (V_{avoid} \times V_{avoid})$.

▶ **Example 8.** Consider the Büchi CG $\mathcal{G}$ from Example 1. Recall that Coverer has a covering strategy in $\mathcal{G}$. We claim that the only forks for β in G are v_1 and v_2. Thus, a winning strategy for Coverer cannot a-priori decompose the objectives in β. Note that this is the case also for the covering strategy described in Example 1.

Indeed, in v_1, Coverer can assign both α_1 and α_2 to an agent that proceeds to u_2 and assign α_3 to an agent that proceeds to d_2. Similarly, in v_2, Coverer can assign α_1 to an agent that proceeds to u_3 and assign both α_2 and α_3 to an agent that proceeds to d_3. Also, as Disruptor may direct tokens to either v_1 or v_2, and the above decompositions are different and are the only possible decompositions in v_1 and v_2, Coverer cannot a-priori decompose β. Hence, v_0 is not a fork. Finally, as β is not covered in each of u_2, d_2, u_3, and d_3, they are not forks either.

Note also that since Disruptor can direct tokens from u_1, m_1, and d_1 back to v_0, the graph G_{avoid} is the sub-graph of G whose vertices are v_0, u_1, m_1 and d_1.

▶ **Theorem 9.** *Consider a CG $\mathcal{G} = \langle G, k, \beta\rangle$, and a vertex $v \in V$. Then, Coverer has a covering strategy in $\langle G^v, k, \beta\rangle$ iff $v \notin V_{avoid}$, or $v \in V_{avoid}$ and Coverer wins $\langle G^v_{avoid}, 1, \beta\rangle$.*

Proof. We first prove that for all vertices $v \notin V_{avoid}$, Coverer has a strategy to cover β from v. Consider a vertex $v \notin V_{avoid}$. By the definition of V_{avoid}, Coverer has a strategy to reach a fork from v. Hence, Coverer can cover β from v by letting all agents follow the same strategy until some (k,l)-fork u for β is reached. By Lemma 7, β is (k,l)-decomposable in u. Hence, once the k tokens of the agents reach u, Coverer can cover β by decomposing it. Let F_1^v denote a covering strategy for Coverer from a vertex $v \notin V_{avoid}$.

We argue that for every vertex $v \in V_{avoid}$, Coverer has a strategy to cover β from v iff Coverer wins $\langle G^v_{avoid}, 1, \beta\rangle$. Consider a vertex $v \in V_{avoid}$, and assume first that Coverer has a covering strategy from v. Since Disruptor can force the play to stay in G_{avoid}, every covering strategy for Coverer from v in G is also a covering strategy from v in G_{avoid}. Also, for every covering strategy $F_1 = \langle f_1^1, \ldots, f_1^k\rangle$ from v, the strategies for the different agents agree with each other as long as the play stays in G_{avoid}. That is, $f_1^1(h) = \cdots = f_1^k(h)$, for every $h \in V^*_{avoid} \cdot V_1'$. Indeed, vertices from which there exists a covering strategy for Coverer that sends different agents to different successors are forks, and by the definition of V_{avoid}, the sub-graph G_{avoid} does not contain forks. Hence, the strategy $f_1 : V^*_{avoid} \cdot V_1' \to V_{avoid}$ with $f_1(h) = f_1^1(h)$ for every $h \in V^*_{avoid} \cdot V_1'$ is a covering strategy for Coverer $\langle G^v_{avoid}, 1, \beta\rangle$.

For the second direction, assume that Coverer wins $\langle G^v_{avoid}, 1, \beta\rangle$. We show that Coverer has a covering strategy from v. Let f_1 be a covering strategy for Coverer in $\langle G^v_{avoid}, 1, \beta\rangle$. Consider the strategy $F_1 = \langle f_1^1, \ldots, f_1^k\rangle$ in which, for all $1 \leq i \leq k$, the strategy f_1^i agrees with f_1 as long as the play stays in G_{avoid}, and proceeds with F_1^u once the play (that is, all tokens together, as this may happen only in a transition taken by Disruptor), leaves G_{avoid} and reaches a vertex $u \notin V_{avoid}$. For every strategy f_2 for Disruptor, either all the outcomes of F_1 and f_2 stay in G_{avoid}, in which case β is covered by each of the agents, or all the outcomes leave G_{avoid}, in which case they follow strategies that cover β by decomposing it. ◀

5 The Complexity of the Coverage Problem

In this section, we study the complexity of deciding the existence of covering strategies in Büchi and co-Büchi CGs, and show that the problem is PSPACE-complete.

We start with upper bounds. Consider a CG $\mathcal{G} = \langle G, k, \beta \rangle$. Recall the set F of forks for β, and the game graph G_{avoid}, in which Disruptor can avoid reaching forks. By Theorem 9, Coverer has a covering strategy in G iff Player 1 has a winning strategy in G_{avoid} for the All objective β, and when the play leaves G_{avoid} (or if the initial vertex is not in G_{avoid}), Coverer can cover β by reaching a fork, from which β can be decomposed.

The above characterization is the key to our algorithm for deciding whether Coverer has a covering strategy. Essentially, the algorithm guesses the set U of forks, checks that Player 1 wins the game with the All objective β in the sub-graph induced by U, and checks (recursively) that the vertices in U are indeed forks. The checks and the maintenance of the recursion require polynomial space. Since NPSPACE=PSPACE, we have the following. [2]

▶ **Theorem 10.** *The coverage problem for Büchi or co-Büchi CGs can be solved in PSPACE.*

Proof. We describe a non-deterministic Turing machine (NTM) T that runs in polynomial space, such that T accepts a CG $\mathcal{G}$ iff Coverer has a covering strategy in $\mathcal{G}$.

Given a CG $\mathcal{G} = \langle G, k, \beta \rangle$, the NTM T guesses a set of vertices $U \subseteq V$, and checks that they are forks. To check that a vertex $u \in U$ is a fork, the NTM guesses a partition $\beta_1, \ldots, \beta_l$ of β and a partition $A_1, \ldots, A_l$ of $[k]$ to nonempty sets, for some $2 \leq l \leq k$, and checks recursively whether Coverer has a covering strategy in $\langle G^u, |A_i|, \beta_i \rangle$, for every $i \in [l]$. If one of the recursive checks fails, the NTM rejects. Otherwise, T calculates the restriction G' of G to vertices from which Disruptor can force the play to avoid U. Then, T accepts if the initial vertex v of G is not in G' or Player 1 wins from v in the All-γ game $\langle G', \beta \rangle$. Otherwise, T rejects.

We prove the correctness of the construction. That is, we prove that T accepts a game $\mathcal{G}$ iff Coverer has a covering strategy in $\mathcal{G}$. By Theorem 9, Coverer has a covering strategy from a vertex v iff $v \notin V_{avoid}$, or $v \in V_{avoid}$ and Coverer wins $\langle G^v_{avoid}, 1, \beta \rangle$. So, if Coverer has a covering strategy, the NTM can guess the set F of forks for the current set β of objectives and number k of agents. Then, G' coincides with G_{avoid}, and by Theorem 9, the NTM accepts. For the second direction, assume the NTM accepts. Then, the vertices in U are forks by definition, and Coverer wins $\langle G', 1, \beta \rangle$. By Theorem 9, it is enough to show that if $v \in G_{avoid}$, then Coverer wins $\langle G^v_{avoid}, 1, \beta \rangle$. Since the vertices in U are forks, G_{avoid} is a sub-graph of G'. So, since Disruptor can force the play to stay in G_{avoid} if v is in it, we also have that Coverer wins $\langle G^v_{avoid}, 1, \beta \rangle$.

To complete the proof, we show that the NTM runs in polynomial space. Note that only a polynomial number of recursive checks can be made before reaching an empty set of objectives, and each check requires a polynomial space for guessing U and the appropriate partitions. Also, calculating G' and checking whether Player 1 wins an All-γ game can be done in polynomial time, for both $\gamma \in \{\text{B,C}\}$ [27]. ◀

Note that the algorithm can return a covering strategy (if one exists). Such a strategy can be described by the set F of forks, the sub-game G_{avoid}, a covering strategy in $\langle G_{avoid}, 1, \beta \rangle$, and the strategies for reaching forks. Then, for each fork, the strategy describes the partitions

[2] It is not hard to see that the proof of Theorem 10 applies to all prefix-independent objective types γ such that All-γ games can be decided in PSPACE.

of β and $[k]$, and the (recursive) description of the strategies in the decomposed game. Note that the description may require more than polynomial space.

We continue to the lower bounds.

▶ **Theorem 11.** *The coverage problem for Büchi or co-Büchi CGs is PSPACE-hard.*

Proof. We describe reductions from QBF. We start with Büchi CGs. Consider a set of variables $X = \{x_1, \ldots, x_n\}$ and a QBF formula $\Phi = Q_1 x_1, \ldots, Q_n x_n \varphi$ where $\varphi = C_1 \wedge \cdots \wedge C_m$, with $C_i = (l_i^1 \vee l_i^2 \vee l_i^3)$, for every $i \in [m]$. We construct a Büchi CG $\mathcal{G} = \langle G, |X|, \beta \rangle$ such that $\Phi = \mathbf{true}$ iff Coverer has a covering strategy in $\mathcal{G}$.

The game graph G (see example in Figure 3) lets the players choose an assignment to the variables in X, following the quantification order of Φ and starting from the outermost variable. Starting from the initial vertex, when the play reaches a vertex that corresponds to an existential variable x_i, Coverer chooses between choosing an assignment to x_i, which involves proceeding to a self-looped sink that corresponds to the literal x_i, or proceeding to a self-looped sink that corresponds to the literal $\overline{x_i}$, and proceeding to the next variables. When the play reaches a vertex that corresponds to a universal variable x_i, Disruptor chooses an assignment to x_i, by proceeding to a vertex that corresponds to the literal x_i, and proceeding to a vertex that corresponds to the literal $\overline{x_i}$. In those vertices, Coverer chooses between staying in the current vertex, and proceeding to the next variables.

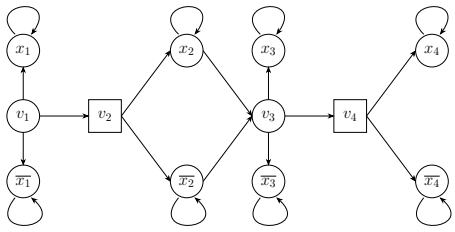


Figure 3 The game graph G for $\Phi = \exists x_1 \forall x_2 \exists x_3 \forall x_4 \varphi$.

Since the vertices that correspond to existential variables are self-looped sinks, the choices of Coverer are not reflected in the history of plays that reach the subsequent variables. Hence, when Disruptor chooses an assignment to a universal variable, she is unaware of the assignments Coverer chose for preceding existential variables. The vertices that correspond to universal variables are not sinks, thus when choosing assignments to existential variables, Coverer is aware of the assignments Disruptor chose for preceding universal variables.

Then, for every variable $x \in X$, we define an objective α_x that forces Coverer to allocate an agent to stay in the vertices that correspond to the assignment chosen for x. Also, for every clause C_i, we define the objective α_{C_i} that consists of vertices that correspond to C_i's literals. Thus, the joint assignment satisfies φ iff Coverer can cover all the objectives. See Appendix A.4 for a formal description of $\mathcal{G}$ and the correctness proof.

For co-Büchi CGs, note that, as in previous reductions, a Büchi objective $\alpha \subseteq (X \cup \overline{X})$ is satisfied in a play ρ iff the co-Büchi objective $(X \cup \overline{X}) \setminus \alpha$ is satisfied in ρ, and thus the same reduction with dual objectives applies. ◀

6 The Complexity of the Disruption Problem

In this section, we study the complexity of deciding the existence of disrupting strategies in Büchi and co-Büchi CGs. Note that since CGs are undetermined, the coverage and disruption problems are not dual, and so the results in Section 5 do not imply that the

disruption problem is PSPACE-complete. In fact, we show here that the disruption problem is Σ_2^{P}-complete. Thus, at least in terms of its theoretical complexity class, it is easier than the coverage problem.

We start with upper bounds. Consider a CG $\mathcal{G} = \langle G, k, \beta \rangle$, and consider a two-player game played on G. For every strategy f_2 for Player 2, let $\Delta_{f_2} \subseteq 2^\beta$ be the set of maximal subsets $\delta \subseteq \beta$ such that Player 1 has a strategy that ensures the satisfaction of the All objective δ when Player 2 uses f_2. That is, Δ_{f_2} is the set of maximal sets $\delta \subseteq \beta$ for which there exists a strategy f_1 for Player 1 such that $\mathsf{sat}(\mathsf{outcome}(\langle f_1, f_2 \rangle), \beta) = \delta$.

▶ **Lemma 12.** *Consider a strategy f_2 for Player 2. For every strategy f_1 for Player 1, there exists $\delta \in \Delta_{f_2}$ such that $\mathsf{outcome}(\langle f_1, f_2 \rangle)$ satisfies the All-$\tilde{\gamma}$ objective $(\beta \setminus \delta)$.*

Proof. Consider a strategy f_2 for Disruptor, and a strategy f_1 for Coverer, and let $\rho = \mathsf{outcome}(\langle f_1, f_2 \rangle)$. By the definition of Δ_{f_2}, there exists $\delta \in \Delta_{f_2}$ such that $\mathsf{sat}(\rho, \beta) \subseteq \delta$. Accordingly, $(\beta \setminus \delta) \cap \mathsf{sat}(\rho, \beta) = \emptyset$, and so ρ satisfies all the $\tilde{\gamma}$ objectives in $\beta \setminus \delta$. ◀

Back to CGs, the intuition behind the definition of Δ_{f_2} is that $\delta \in \Delta_{f_2}$ iff whenever Disruptor follows f_2, Coverer can allocate one agent that covers exactly all the objectives in δ. Formally, we have the following, which follows directly from the definitions.

▶ **Lemma 13.** *Consider a CG $\mathcal{G} = \langle G, k, \beta \rangle$. A strategy f_2 for Disruptor is a disrupting strategy in $\mathcal{G}$ iff for every k sets $\delta_1, \ldots, \delta_k \in \Delta_{f_2}$, we have that $\bigcup_{i \in [k]} \delta_i \neq \beta$.*

We use the observations about Δ_{f_2} in order to restrict the search for disrupting strategies. We start with Büchi CGs.

▶ **Theorem 14.** *Disruptor has a disrupting strategy in a Büchi CG $\mathcal{G}$ iff she has a memoryless disrupting strategy in $\mathcal{G}$.*

Proof. Consider a Büchi CG $\mathcal{G} = \langle G, k, \beta \rangle$. Assume Disruptor has a disrupting strategy f_2 in $\mathcal{G}$. By Lemma 12, for every strategy f_1 for an agent of Coverer, there exists $\delta \in \Delta_{f_2}$ such that $\mathsf{outcome}(\langle f_1, f_2 \rangle)$ satisfies the AllC objective $\beta \setminus \delta$, which is equivalent to the co-Büchi objective $\cup(\beta \setminus \delta)$. Accordingly, f_2 is a winning strategy for Player 2 in a game played on G in which her objective is the ExistsC objective $\{\cup(\beta \setminus \delta)\}_{\delta \in \Delta_{f_2}}$. Let f_2' be a memoryless winning strategy for Player 2 in this game. Since ExistsC objectives require memoryless strategies, such a strategy exists. In Appendix A.5, we prove that f_2' is a disrupting strategy in $\mathcal{G}$. ◀

▶ **Theorem 15.** *The disruption problem for Büchi CGs is in Σ_2^{P}.*

Proof. Consider a Büchi CG $\mathcal{G} = \langle G, k, \beta \rangle$. An NP algorithm that uses a co-NP oracle guesses a memoryless strategy f_2 for Disruptor, and then checks that Coverer does not have a covering strategy in the one-player game defined over the sub-graph G_{f_2} induced by f_2. That is, the sub-graph of G in which the edges from vertices in V_2 agree with f_2. Since Coverer essentially owns all the vertices in G_{f_2}, then by Theorem 2, deciding the existence of covering strategies in G_{f_2} can be done in NP, implying the Σ_2^{P}-upper bound. ◀

We continue to co-Büchi CGs. Here, disrupting strategies need not be memoryless, and in fact need not be polynomial. Accordingly, the proof is more complicated and is based on a symbolic description of disrupting strategies.

A *superset objective* is a set $\mathcal{F} \subseteq 2^{2^V}$ of AllB objectives, and it is satisfied in a play ρ iff at least one AllB objective in $\mathcal{F}$ is satisfied in ρ [16]. By [16], every superset objective has an equivalent AllB objective. Accordingly, in two-player superset games, Player 2 has

an ExistsC objective, thus Player 2 wins iff she has a memoryless winning strategy. Since a memoryless strategy for Player 2 can be checked in polynomial time and the game is determined, the problem of deciding whether there exists a winning strategy for a superset objective is co-NP-complete [16].

We start with an easy characterization of winning strategies for AllB objectives.

▶ **Lemma 16.** *Consider an AllB game $\mathcal{G} = \langle G, \delta \rangle$, with $\delta = \{\alpha_1, \ldots, \alpha_m\}$. Player 1 wins $\mathcal{G}$ iff there exist a set $W \subseteq V$ of vertices and a set $\{g_1^i : i \in [m]\}$ of memoryless strategies for Player 1, such that Player 1 can force the play to reach W, and for every vertex $v \in W$ and $i \in [m]$, the strategy g_1^i forces each play from v to reach $\alpha_i \cap W$, while staying in W.*

We say that W is a *winning cage* for δ, and that $\{g_1^i : i \in [m]\}$ are its *witnesses*.

Lemma 17 below suggests a way to describe winning strategies for AllB objectives by the sets of successors a winning strategy chooses infinitely often from every vertex.

For a set of vertices $U \subseteq V$, a strategy f_1 for Player 1 is a *U-trap* if it forces the play to reach and stay in U. That is, if $\mathit{inf}(\mathsf{outcome}(\langle f_1, f_2 \rangle)) \subseteq U$, for every strategy f_2 for Player 2.

A *fairness requirement* for Player 1 is a function $g : V_1 \to 2^V$ that maps each vertex $v \in V_1$ to a subset of its successors. We say that a strategy f_1 for Player 1 *respects* g if for every vertex $v \in V_1$, the strategy f_1 only chooses successors for v from $g(v)$, and when v is visited infinitely often, then f_1 proceeds to each of the successors of v in $g(v)$ infinitely often. That is, for every vertex $v \in V_1$ and a prefix of a play $h \in V^*$ we have that $f_1(h \cdot v) \in g(v)$, and for every strategy f_2 for Player 2 and a vertex $v \in V_1$, if $v \in \mathit{inf}(\mathsf{outcome}(\langle f_1, f_2 \rangle))$, then $g(v) \subseteq \mathit{inf}(\mathsf{outcome}(\langle f_1, f_2 \rangle))$.

▶ **Lemma 17.** *Consider an AllB game $\mathcal{G} = \langle G, \delta \rangle$, with $\delta = \{\alpha_1, \ldots, \alpha_m\}$. Player 1 wins $\mathcal{G}$ iff there exists a set of vertices $U \subseteq V$ and a fairness requirement g for Player 1 such that every U-trap strategy for Player 1 that respects g is a winning strategy for Player 1 in $\mathcal{G}$, and such a strategy exists.*

Proof. First, if there is a U-trap strategy for Player 1 that respects a fairness requirement g such that every such strategy is a winning strategy in $\mathcal{G}$, then Player 1 has a winning strategy in $\mathcal{G}$.

For the other direction, assume Player 1 wins $\mathcal{G}$. Consider a minimal winning cage $W \subseteq V$ and witnesses $\{g_1^i : i \in [m]\}$ for W. Note that by definition, Player 1 has a W-trap strategy that respects a fairness requirement g with $g(v) = \{g_1^i(v) : i \in [m]\}$ for every $v \in V_1 \cap W$. We show that every such strategy is a winning strategy for Player 1 in $\mathcal{G}$.

Consider a strategy f_1^* for Player 1 in $\mathcal{G}$ that forces the play to reach W, and, starting with $i = 1$, follows g_1^i until the play reaches α_i, where it switches to follow $g_1^{(i+1) \mod m}$. It is easy to see that f_1^* is a winning strategy for Player 1 in $\mathcal{G}$.

Now, consider a strategy f_1 for Player 1 that is a W-trap, and respects g. Note that such a strategy exists, since W is a winning cage for δ and for every vertex $v \in V_1 \cap W$, we have that $g(v) \subseteq W$. We show that f_1 is a winning strategy for Player 1 in $\mathcal{G}$.

Assume by contradiction otherwise. Thus, there exists a strategy f_2 for Player 2 such that $U = \mathit{inf}(\mathsf{outcome}(\langle f_1, f_2 \rangle))$ does not intersect with all the sets in δ. That is, there exists $i \in [m]$ such that $U \cap \alpha_i = \emptyset$. We show that there exists a strategy f_2' for Player 2 with $U' = \mathit{inf}(\mathsf{outcome}(\langle f_1^*, f_2' \rangle))$ such that $U' \subseteq U$, thus $U' \cap \alpha_i = \emptyset$, which contradicts the fact that f_1^* is a winning strategy for Player 1 in $\mathcal{G}$.

Let f_2' be a strategy for Player 2 with which $\mathsf{outcome}(\langle f_1^*, f_2' \rangle)$ reaches U, and chooses for every vertex $v \in U \cap V_2$ a successor in U. Note that Player 2 can make the play reach U when Player 1 uses f_1^*, as otherwise f_1^* is a strategy that forces the play to avoid U, which

contradicts the fact W is minimal. Note that $U' \subseteq U$, since both strategies choose successors in U for vertices in U. Indeed, by the definition of f_1^*, for every vertex $v \in U' \cap V_1$ the set of successors f_1^* chooses for v is a subset of $g(v) = \{g_1^i(v) : i \in [m]\}$. To conclude, note that by the definition of f_1, we have that $g(v) \subseteq U$. Indeed, every vertex $v \in U \cap V_1$ is visited infinitely often in $\mathsf{outcome}(\langle f_1, f_2\rangle)$, thus all the vertices in $g(v)$ are visited infinitely often. ◀

By Lemma 17, we do not need to guess an explicit strategy for Player 1 in order to determine whether she wins an AllB or a superset game. Instead, we can guess a set of vertices that might appear infinitely often, and the set of successors we choose infinitely often from each of those vertices.

Back to CGs, we show that disrupting strategies can be described symbolically using a fairness requirement.

We say that a set of subsets of objectives $\Delta \subseteq 2^\beta$ is *k-wise intersecting* if the intersection of every k sets in Δ is not empty. That is, $\bigcap_{i\in[k]} \delta_i \neq \emptyset$, for every $\delta_1, \ldots, \delta_k \in \Delta$.

▶ **Lemma 18.** *A strategy f_2 for Disruptor is a disrupting strategy iff the set $\{(\beta \setminus \delta) : \delta \in \Delta_{f_2}\}$ is k-wise intersecting.*

Proof. Assume first that f_2 is a disrupting strategy, and consider k sets $\delta_1, \ldots, \delta_k \in \Delta_{f_2}$. By the definition of Δ_{f_2}, there exist strategies $f_1^1, \ldots, f_1^k$ for the agents of Coverer with $\mathsf{sat}(\mathsf{outcome}(\langle f_1^i, f_2\rangle), \beta) = \delta_i$, for every $i \in [k]$. Since f_2 is a disrupting strategy, then, by Lemma 13, we have that $\bigcup_{i\in[k]} \delta_i \neq \beta$. Therefore, $\bigcap_{i\in[k]} (\beta \setminus \delta_i) \neq \emptyset$, and so $\{(\beta \setminus \delta) : \delta \in \Delta_{f_2}\}$ is k-wise intersecting.

For the second direction, consider a strategy f_2 for Disruptor such that $\{(\beta \setminus \delta) : \delta \in \Delta_{f_2}\}$ is k-wise intersecting. Then, for every strategy $F_1 = \langle f_1^1, \ldots, f_1^k\rangle$ for Coverer and $\delta_1, \ldots, \delta_k \in \Delta_{f_2}$ with $\mathsf{sat}(\mathsf{outcome}(f_1^i, f_2)) \subseteq \delta_i$, for every $i \in [k]$, there exists $\alpha_j \in \bigcap_{i\in[k]} (\beta \setminus \delta_i)$. Thus, α_j is not covered in $\mathsf{outcome}(\langle F_1, f_2\rangle)$. Accordingly, f_2 is a disrupting strategy. ◀

▶ **Lemma 19.** *If Disruptor has a disrupting strategy in a co-Büchi CG, then there exist a set of vertices U and a fairness requirement g such that every U-trap strategy for Disruptor that respects g is a disrupting strategy, and such a strategy exists.*

Proof. Consider a co-Büchi CG $\mathcal{G} = \langle G, k, \beta\rangle$, and assume there exists a disrupting strategy f_2 for Disruptor. Let δ_{f_2} be the AllB objective equivalent to the superset objective $\{(\beta \setminus \delta) : \delta \in \Delta_{f_2}\}$. Since f_2 is a winning strategy for the superset objective $\{(\beta \setminus \delta) : \delta \in \Delta_{f_2}\}$, and thus for its equivalent AllB objective δ_{f_2}, then, by Lemma 17, there exist a set of vertices U and a fairness requirement g such that every U-trap strategy f_2' for Disruptor that respects g is a winning strategy for δ_{f_2}, and hence also a winning strategy for the superset objective $\{(\beta \setminus \delta) : \delta \in \Delta_{f_2}\}$. Also, such a strategy exists. By Lemma 18, in order to show that every such strategy f_2' is a disrupting strategy, it is enough to show that $\{(\beta \setminus \delta) : \delta \in \Delta_{f_2'}\}$ is k-wise intersecting.

Consider a U-trap strategy f_2' that respects g. Note that since f_2' is a winning strategy for the superset objective $\{(\beta \setminus \delta) : \delta \in \Delta_{f_2}\}$, for every set $\delta \in \Delta_{f_2'}$, there exists a set $\delta' \in \Delta_{f_2}$ such that $(\beta \setminus \delta') \subseteq (\beta \setminus \delta)$. Now, consider k sets $\delta_1, \ldots, \delta_k \in \Delta_{f_2'}$ of objectives, and for every $i \in [k]$, let $\delta_i' \in \Delta_{f_2}$ be a set of objectives such that $(\beta \setminus \delta_i') \subseteq (\beta \setminus \delta_i)$. Since $\bigcap_{i\in[k]} (\beta \setminus \delta_i') \subseteq \bigcap_{i\in[k]} (\beta \setminus \delta_i)$ and $\{(\beta \setminus \delta) : \delta \in \Delta_{f_2}\}$ is k-wise intersecting, we have that $\bigcap_{i\in[k]} (\beta \setminus \delta_i) \neq \emptyset$. Accordingly, $\{(\beta \setminus \delta) : \delta \in \Delta_{f_2'}\}$ is k-wise intersecting. ◀

The fact we can restrict attention to disrupting strategies that are given by fairness requirements leads to an upper bound for the disruption problem.

▶ **Theorem 20.** *The disruption problem for co-Büchi CGs is in* Σ_2^{P}.

Proof. An NP algorithm that uses a co-NP oracle guesses a set of vertices $U \subseteq V$ and a fairness requirement $g : (V_2 \cap U) \to (2^U \setminus \{\emptyset\})$, checks that there exists a U-trap strategy for Disruptor that respects g, and then the co-NP oracle checks that every U-trap strategy for Disruptor that respects g is a disrupting strategy. By Lemma 19, the search can be restricted to trap strategies that respect a given fairness requirement. In Appendix A.6 we describe how the above two steps can indeed be done in NP and co-NP. In particular, checking the U-traps that respect g is based on the fact that for every strategy for an agent of Coverer, the set of vertices visited infinitely often in the resulted play are strongly connected subsets of U that contain all the successors of vertices in V_2 according to g, and the rest of the vertices are successors of vertices in V_1. ◀

We continue to lower bounds. Note that the game in the reduction in the proof of Theorem 11 is determined when Φ is of the form $\forall X \exists Y \varphi$. Indeed, in this case the assignment to the universal variables is independent of the assignment to the existential variables, and thus if $\Phi =$ **false**, which holds iff the 2QBF formula $\exists X \forall Y \overline{\varphi}$ is valid, Disruptor has a disrupting strategy. Since 2QBF is Σ_2^{P}-hard, we have the following (see full proof in Appendix A.7).

▶ **Theorem 21.** *The disruption problem for Büchi or co-Büchi CGs is* Σ_2^{P}*-hard.*

7 Discussion

We introduced and studied coverage games, extending multi-agent planning to adversarial factors that go beyond those studied in existing frameworks.

Below we discuss variants of CGs. Clearly, all classical extensions to the underlying two-player game (e.g., concurrency, probability, partial visibility, etc.) and to the objectives (e.g., richer winning conditions, weighted objectives, mixing objectives of different types, etc.), as well as classical extensions from planning (e.g., dynamic objectives, optimality of agents and their resources, etc.) are interesting also in the setting of CGs. For example, handling *reachability* objectives, it is interesting to search for strategies of Coverer that reach the objectives via short paths, and one may optimize the longest or the average path. Although the extension to richer objectives is an obvious direction for future work, we note that the study of Büchi and co-Büchi better highlights the differences between CGs and standard games, as the complexity is not dominated by challenges that have to do with the objective. For example, by Footnote 2, the extension of the general Müller objective to CGs involves no cost, whereas for the Büchi and co-Büchi objectives, we exhibit a variety of complexity classes, obtained for the different variants of CGs. As our results show, the transition from multiple objectives (a single agent) to CGs adds the challenge of decomposing the set β of objectives among the agents. For example, while multiple co-Büchi objectives can be merged, making them easier than multiple Büchi objectives, co-Büchi CGs are not easier than Büchi CGs. It is interesting to study how the need to decompose the objectives affects the problem in richer settings.

More interesting in our context are extensions that have to do with the operation of several agents: communication among the agents and the ability of Disruptor to also use different agents. Recall that in the CGs studied here, the strategy of each agent may depend on the history of her interaction with Disruptor, but is independent of the interaction of the other agents. Settings in which the agents communicate with each other corresponds to CGs in which their strategies depend on the history of all interactions. Clearly, communication

can help Coverer. For example, if two drones are tasked with patrolling an area and Drone 1 is drifting northward at a certain location, knowing this could prompt Drone 2 to avoid that location or to fly southward. Also, beyond full information about the other agents, various applications induce interesting special cases, such as visibility of a subset of the agents, of agents in some radius, of their current location only, and so on.

As for a richer disruption, recall that in the CGs studied here, all the agents of Coverer interact with the same strategy of Disruptor. It is interesting to consider settings in which Disruptor also operates a number l of agents. Then, a strategy for Disruptor is a vector $F_2 = \langle f_2^1, \ldots, f_2^l \rangle$ of strategies, and Coverer should cover all the objectives in β for every F_2 and for every possible pairing of each of her agents with those of Disruptor. This extension corresponds to settings in which different agents may face different responses, even after the exact same interaction. Clearly, this can help Disruptor. For example, if a system aims to ensure that some road remains uncongested, it may limit the number of cars in each zone by directing different cars to different directions. From a technical point of view, the setting involves additional challenges. In particular, decomposition now has an additional parameter. For example, it can be shown that if $l = k$, then Coverer can cover β iff there is an a-priori decomposition of β among the agents.

References

1 L. Alon, N. Agmon, and G.A Kaminka. Taking turns in complete coverage for multiple robots. In *Distributed Autonomous Robotic Systems*, pages 401–412. Springer International Publishing, 2019.

2 N. Bertrand, M. Dewaskar, B. Genest, H. Gimbert, and A.A. Godbole. Controlling a population. *Logical Methods in Computer Science*, 15(3):1–30, 2019. doi:10.23638/LMCS-15(3:6)2019.

3 R. Bloem, K. Chatterjee, and B. Jobstmann. Graph games and reactive synthesis. In *Handbook of Model Checking.*, pages 921–962. Springer, 2018. doi:10.1007/978-3-319-10575-8_27.

4 P. Bouyer, R. Brenguier, N. Markey, and M. Ummels. Concurrent games with ordered objectives. In *Proc. 15th Int. Conf. on Foundations of Software Science and Computation Structures*, volume 7213 of *Lecture Notes in Computer Science*, pages 301–315. Springer, 2012. doi:10.1007/978-3-642-28729-9_20.

5 K. Chatterjee, L. Doyen, T.A. Henzinger, and J-F. Raskin. Generalized mean-payoff and energy games. In *Proc. 30th Conf. on Foundations of Software Technology and Theoretical Computer Science*, volume 8 of *LIPIcs*, pages 505–516. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2010. doi:10.4230/LIPICS.FSTTCS.2010.505.

6 K. Chatterjee, W. Dvorak, M. Henzinger, and V. Loitzenbauer. Conditionally optimal algorithms for generalized büchi games. In *41st Int. Symp. on Mathematical Foundations of Computer Science*, volume 58 of *LIPIcs*, pages 25:1–25:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPICS.MFCS.2016.25.

7 K. Chatterjee, T. A. Henzinger, and N. Piterman. Generalized parity games. In *Proc. 10th Int. Conf. on Foundations of Software Science and Computation Structures*, volume 4423 of *Lecture Notes in Computer Science*, pages 153–167. Springer, 2007. doi:10.1007/978-3-540-71389-0_12.

8 K. Chatterjee, Z. Komárková, and J. Kretínský. Unifying two views on multiple mean-payoff objectives in markov decision processes. In *Proc. 30th ACM/IEEE Symp. on Logic in Computer Science*, pages 244–256, 2015.

9 S.W. Cho, H.J. Park, H. Lee, D.H. Shim, and S.Y. Kim. Coverage path planning for multiple unmanned aerial vehicles in maritime search and rescue operations. *Computers and Industrial Engineering*, 161:107612, 2021. doi:10.1016/j.cie.2021.107612.

10 H. Choset. Coverage for robotics - a survey of recent results. *Annals of Mathematics and Artificial Intelligence*, 31:113–126, 2001. doi:10.1023/A:1016639210559.

11 A.G. Clark, N. Walkinshaw, and R.M. Hierons. Test case generation for agent-based models: A systematic literature review. *Information and Software Technology*, 135:106567, 2021. doi:10.1016/J.INFSOF.2021.106567.

12 T. Colcombet, N. Fijalkow, and P. Ohlmann. Controlling a random population. *Logical Methods in Computer Science*, Volume 17, Issue 4, November 2021.

13 N. Fijalkow and F. Horn. The surprizing complexity of reachability games. *CoRR*, abs/1010.2420, 2010. arXiv:1010.2420.

14 H. Gimbert, C. Mascle, and P. Totzke. Optimally controlling a random population, 2024. arXiv:2411.15181.

15 D. Harel and A. Pnueli. On the development of reactive systems. In K. Apt, editor, *Logics and Models of Concurrent Systems*, volume F-13 of *NATO Advanced Summer Institutes*, pages 477–498. Springer, 1985.

16 P. Hunter and A. Dawar. Complexity bounds for regular games. In *30th Int. Symp. on Mathematical Foundations of Computer Science*, volume 3618, pages 495–506. Springer, 2005. doi:10.1007/11549345_43.

17 O. Keidar and N. Agmon. Safe navigation in adversarial environments. *Annals of Mathematics and Artificial Intelligence*, 83(2):121–164, 2018. doi:10.1007/S10472-018-9591-0.

18 O. Kupferman, N. Piterman, and M.Y. Vardi. Safraless compositional synthesis. In *Proc. 18th Int. Conf. on Computer Aided Verification*, volume 4144 of *Lecture Notes in Computer Science*, pages 31–44. Springer, 2006. doi:10.1007/11817963_6.

19 O. Kupferman and N. Shenwald. Games with weighted multiple objectives. In *22nd Int. Symp. on Automated Technology for Verification and Analysis*, Lecture Notes in Computer Science. Springer, 2024.

20 E. Lin, N. Agmon, and S. Kraus. Multi-robot adversarial patrolling: Handling sequential attacks. *Artificial Intelligence*, 274, 2019.

21 D.A. Martin. Borel determinacy. *Annals of Mathematics*, 65:363–371, 1975.

22 N. Piterman, A. Pnueli, and Y. Saar. Synthesis of reactive(1) designs. In *Proc. 7th Int. Conf. on Verification, Model Checking, and Abstract Interpretation*, volume 3855 of *Lecture Notes in Computer Science*, pages 364–380. Springer, 2006. doi:10.1007/11609773_24.

23 A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proc. 16th ACM Symp. on Principles of Programming Languages*, pages 179–190, 1989.

24 K.S. Senthilkumar and K.K. Bharadwaj. Multi-robot exploration and terrain coverage in an unknown environment. *Robotics and Autonomous Systems*, 60(1):123–132, 2012. doi:10.1016/J.ROBOT.2011.09.005.

25 C.S. Tan, R. Mohd-Mokhtar, and M.R. Arshad. A comprehensive review of coverage path planning in robotics using classical and heuristic algorithms. *IEEE Access*, 9:119310–119342, 2021. doi:10.1109/ACCESS.2021.3108177.

26 E.I. Tolstaya, J. Paulos, V. Kumar, and A. Ribeiro. Multi-robot coverage and exploration using spatial graph neural networks. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 8944–8950, 2021.

27 M.Y. Vardi and P. Wolper. Automata-theoretic techniques for modal logics of programs. *Journal of Computer and Systems Science*, 32(2):182–221, 1986. doi:10.1016/0022-0000(86)90026-7.

28 M.Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, 1994. doi:10.1006/INCO.1994.1092.

29 Y. Velner, K. Chatterjee, L. Doyen, T.A. Henzinger, A. Rabinovich, and J-F. Raskin. The complexity of multi-mean-payoff and multi-energy games. *Information and Computation*, 241:177–196, 2015. doi:10.1016/J.IC.2015.03.001.

30 W. Zielonka. Infinite games on finitely coloured graphs with applications to automata on infinite trees. *Theoretical Computer Science*, 200(1-2):135–183, 1998. doi:10.1016/S0304-3975(98)00009-7.

A Missing proofs

A.1 Missing details in the proof of Theorem 2

We start with the case $\gamma = \text{B}$. There, $\mathcal{G} = \langle G', k, E\rangle$, where the game graph $G' = \langle V', \emptyset, v_0, E'\rangle$ has the following components.

1. $V' = V \cup \{v_0\}$, where $v_0 \notin V$.
2. The set of edges E' contains the following edges.
   a. $\langle v_0, v\rangle$, for every $v \in V$. That is, from the initial vertex, Coverer chooses a vertex $v \in V$ by proceeding to the vertex that correspond to v in V'.
   b. $\langle v, v\rangle$, for every $v \in V$.

We prove the correctness of the reduction. Assume first that there exists a vertex-cover $U \subseteq V$ of size k in G. Then, the strategy F_1 for Coverer in $\mathcal{G}$ that sends a different agent to every vertex $v \in U$ is a covering strategy. Indeed, since U is a vertex cover, for every edge $e = \{u, v\} \in E$ we have that $u \in U$ or $v \in U$, thus the Büchi objective e is satisfied in the play in $\mathsf{outcome}(F_1)$ that reaches the self-looped sink v or the self-looped sink u.

For the second direction, assume there exists a covering strategy F_1 in $\mathcal{G}$. Let U be the set of k self-looped sinks that the plays in $\mathsf{outcome}(F_1)$ reach. Then, U is a vertex-cover in G of size k. Indeed, for every edge $e = \{u, v\} \in E$, if every play in $\mathsf{outcome}(F_1)$ does not reach u and does not reach v, then there does not exist a play in $\mathsf{outcome}(F_1)$ that satisfies the Büchi objective e, which is not possible since F_1 is a covering strategy.

For $\gamma = \text{C}$, we define the same graph G, with $\beta = \{V \setminus e : e \in E\}$. Since each play ρ is eventually trapped in a self-loop in a vertex in V, the co-Büchi objective $V \setminus e$ is satisfied in ρ iff the Büchi objective e is satisfied in ρ, thus the game coincides with the one for $\gamma = \text{B}$.

A.2 Missing details in the proof of Theorem 5

We first prove that every strategy $F_1 = \langle f_1^1, f_1^2\rangle$ for Coverer is not a covering strategy in $\mathcal{G}$. If $f_1^1(v_0) = f_1^2(v_0) = v_1$, then F_1 satisfies at most α_1 and α_2; If $f_1^1(v_0) = f_1^2(v_0) = v_2$, then F_1 guarantees the satisfaction of the two objectives α_1 and α_3, when Disruptor proceeds from v_2 to u_2, and the two objectives α_2 and α_3, when Disruptor proceeds from v_2 to d_2; Otherwise, without loss of generality, we have that $f_1^1(v_0) = v_1$ and $f_1^2(v_0) = v_2$. If $f_1^1(v_1) = u_1$, then when Disruptor proceeds from v_2 to u_2, only α_1 and α_3 are satisfied, and in a similar way, if $f_1^1(v_1) = d_1$, then when Disruptor proceeds from v_2 to d_2, only α_2 and α_3 are satisfied. Therefore, F_1 is not a covering strategy.

We now prove that every strategy f_2 for Disruptor is not a disrupting strategy. If $f_2(v_2) = u_2$, then when Coverer uses the strategy in which one agent goes to d_1 and one agent goes to v_2, all three objectives are satisfied; in a similar way, if $f_2(v_2) = d_2$, then when Coverer uses the strategy in which one agent goes to u_1 and one agent goes to v_2, all three objectives are satisfied. Therefore, f_2 is not a disrupting strategy.

A.3 Proof of Theorem 6

For the first claim, consider a CG $\mathcal{G} = \langle G, k, \beta\rangle$, and consider $1 < l \leq k$ and a vertex v of G such that β is (k, l)-decomposable in v. We show that β is also $(k, l-1)$-decomposable in v. The claim for all $l' \leq l$ then follows by repeated (possibly zero) applications of this claim. Since β is (k, l)-decomposable in v, there exists a partition $\beta_1, \ldots, \beta_l$ of β and a partition $A_1, \ldots, A_l$ of A to nonempty sets, such that for every $1 \leq i \leq l$, Coverer wins the game $\langle G^v, |A_i|, \beta_i\rangle$. It follows that Coverer also wins the CG $\langle G^v, |A_{l-1} \cup A_l|, \beta_{l-1} \cup \beta_l\rangle$.

Indeed, the strategy in which the agents in A_{l-1} use their winning strategy for β_{l-1} and the agents in A_l use their winning strategy for β_l covers $\beta_{l-1} \cup \beta_l$. Hence, $\beta_1, \ldots, \beta_{l-2}, \beta_{l-1} \cup \beta_l$ and $A_1, \ldots, A_{l-2}, A_{l-1} \cup A_l$ are partitions of β and A, respectively, to $l-1$ nonempty sets, witnessing that β is $(k, l-1)$-decomposable from v.

For the second item, consider $k \geq 2$. We prove the claim for $l = 2$. By the first item, the result then follows for all $l > 1$. First, for $\gamma = B$, consider the γ-CG $\mathcal{G} = \langle G, k, \beta \rangle$, where G is as follows (see Figure 4 for an example for $k = 2$). Starting from the initial vertex v_0, Disruptor chooses between $k+1$ vertices $v_1, \ldots, v_{k+1}$. From v_i, Coverer chooses among k self-looped sinks $s_i^1, \ldots, s_i^k$, for every $i \in [k+1]$. Then, for every $i \in [k+1]$ and $j \in [i-1]$, the sink s_i^j satisfies the objective α_j, and for every $j \in [k+1] \setminus [i]$, the sink s_i^{j-1} satisfies the objective α_j. The sink s_i^1 also satisfies α_i. That is, every objective $\alpha_j \in \beta \setminus \{\alpha_i\}$ is satisfied in a separate sink reachable from v_i. Formally, $\beta = \{\alpha_i : i \in [k+1]\}$, with $\alpha_i = \{s_j^i : i+1 \leq j \leq k+1\} \cup \{s_j^{i-1} : 1 \leq j \leq i-1\} \cup \{s_i^1\}$, for every $i \in [k+1]$.

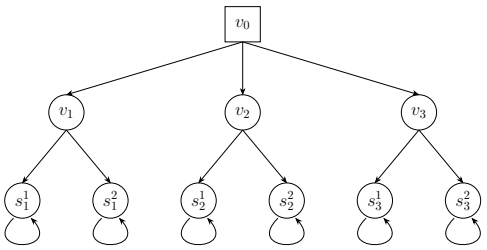


Figure 4 The game graph G for $k = 2$. Here, $\beta = \{\alpha_1, \alpha_2, \alpha_3\}$, with $\alpha_1 = \{s_2^1, s_3^1, s_1^1\}$, $\alpha_2 = \{s_3^2, s_1^1, s_2^1\}$, and $\alpha_3 = \{s_1^2, s_2^2, s_3^1\}$.

It is not hard to see that Coverer wins $\mathcal{G}$. Indeed, β can be satisfied in G^{v_i}, for every successor v_i of v_0. We show that β is not $(k, 2)$-decomposable in G. Assume by contradiction otherwise. Thus, there exist $\beta' \subset \beta$ and $A' \subset A$ such that Coverer wins $\langle G, |A'|, \beta' \rangle$ and $\langle G, |A \setminus A'|, \beta \setminus \beta' \rangle$. Note that every winning strategy for Coverer in $\langle G, |A'|, \beta' \rangle$ requires $|\beta'|$ agents. Indeed, since $|\beta'| \leq k$, Disruptor can choose from the initial vertex a successor v_i such that $\alpha_i \notin \beta'$, thus β' is covered from v_i iff $|\beta'|$ agents are sent to $|\beta'|$ different successors of v_i. In a similar way, every winning strategy for Coverer in $\langle G, |A \setminus A'|, \beta \setminus \beta' \rangle$ requires $|\beta \setminus \beta'|$ agents. Therefore, $k = |A| = |A'| + |A \setminus A'| \geq |\beta'| + |\beta \setminus \beta'| = |\beta| = k+1$, contradiction.

Next, for $\gamma = C$, we define $\mathcal{G}' = \langle G, k, \beta' \rangle$ over the same game graph, now with $\beta' = \{\alpha'_1, \ldots, \alpha'_{k+1}\}$, where $\alpha'_i = \cup(\beta \setminus \{\alpha_i\})$, for every $i \in [k+1]$. Since the vertices s_i^j are sinks, every play in G satisfies the γ objective α'_i iff it satisfies the $\tilde{\gamma}$ objective α_i, for every $i \in [k+1]$, we have that Coverer wins $\mathcal{G}'$ but β' is not $(k, 2)$-decomposable in G.

We continue to the last claim. Consider $k \geq 2$ and $1 \leq l < k$. First, for $\gamma = B$, we describe a γ-CG $\mathcal{G} = \langle G, k, \beta \rangle$ with $\beta = \{\alpha_1, \ldots, \alpha_{k+1}\}$ such that β is (k, l)-decomposable in G, but β is not $(k, l+1)$-decomposable in G. The game is as follows. Let $\mathcal{G}' = \langle G', k-(l-1), \beta' \rangle$ be a CG with $\beta' = \{\alpha'_1, \ldots, \alpha'_{k-(l-1)+1}\}$ such that Coverer wins $\mathcal{G}'$, but β' is not $(k-(l-1), 2)$-decomposable in G'. By item 2, such a CG exists. Starting from the initial vertex in G, Disruptor chooses between l successors $v_1, \ldots, v_l$. The vertices $v_1, \ldots, v_{l-1}$ are self-looped sinks, and from v_l, Coverer proceeds to the sub-graph G'. Then, we define $\alpha_i = \{v_i\}$, for every $i \in [l-1]$, and $\alpha_i = \alpha'_{i-(l-1)}$, for every $l \leq i \leq k+1$. It is not hard to see that β is (k, l)-decomposable in G. Indeed, for every $i \in [l-1]$, Coverer wins $\langle G, 1, \{\alpha_i\} \rangle$, and also Coverer wins $\langle G, k-(l-1), \beta' \rangle$, since Coverer wins $\mathcal{G}'$. We continue to show that β is not $(k, l+1)$-decomposable. Assume by contradiction otherwise, and let $\beta_1, \ldots, \beta_{l+1}$ and $A_1, \ldots, A_{l+1}$ be partitions of β and A to $l+1$ nonempty sets, respectively, such that Coverer

wins $\langle G, |A_i|, \beta_i\rangle$, for every $1 \leq i \leq l+1$. Since there are only $l-1$ objectives in $\beta \setminus \beta'$, we have that β' is divided among at least two different sets in the partition, implying an a-priori partition of β', contradicting the fact β' is not $(k-(l-1),2)$-decomposable in G'. The same results hold for $\gamma = C$ and the γ-CG $\mathcal{G}' = \langle G, k, \beta''\rangle$ with $\beta'' = \{\alpha''_1, \ldots, \alpha''_{k+1}\}$ such that $\alpha''_i = \cup(\beta \setminus \{\alpha_i\})$, for every $i \in [k+1]$.

A.4 Missing details in the proof of Theorem 11

Formally, $\mathcal{G} = \langle G, |X|, \beta\rangle$, where the game graph $G = \langle V_1, V_2, v_1, E\rangle$ has the following components.

1. $V_1 = \{v_i : i \in [n] \text{ and } Q_i = \exists\} \cup \{l : l \in X \cup \overline{X}\}$. The vertices in $\{v_i : i \in [n] \text{ and } Q_i = \exists\}$ are *existential variable vertices*, and the vertices in $\{l : l \in X \cup \overline{X}\}$ are *literal vertices*.
2. $V_2 = \{v_i : i \in [n] \text{ and } Q_i = \forall\}$. The vertices in $\{v_i : i \in [n] \text{ and } Q_i = \forall\}$ are *universal variable vertices*.
3. The set E of edges includes the following edges.
   a. $\langle v_i, x_i\rangle$ and $\langle v_i, \overline{x_i}\rangle$, for every $i \in [n]$. That is, from the variable vertex v_i, the owner of the vertex chooses an assignment to the variable v_i by proceeding to one of the literal vertices x_i and $\overline{x_i}$, which correspond to assigning **true** and **false**, respectively.
   b. $\langle l, l\rangle$, for every $l \in X \cup \overline{X}$. That is, Coverer can choose to stay in every literal vertex.
   c. $\langle v_i, v_{i+1}\rangle$, for every $i \in [n-1]$ such that $Q_i = \exists$. That is, from existential variable vertices, Coverer can proceed to the next variable vertex.
   d. $\langle l, v_{i+1}\rangle$, for every $i \in [n-1]$ such that $Q_i = \forall$ and $l \in \{x_i, \overline{x_i}\}$. That is, from literal vertices that correspond to universal variables, Coverer can choose between staying in the literal vertex, and proceeding to the next variable vertex.

The set β contains the following objectives.

1. $\alpha_x = \{x, \overline{x}\}$, for every $x \in X$.
2. $\alpha_{C_i} = \{l_i^1, l_i^2, l_i^3\}$, for every $i \in [m]$.

We prove the correctness of the construction. That is, we prove that $\Phi =$ **true** iff Coverer has a covering strategy in $\mathcal{G}$. Note that the objective α_x, for every $x \in X$, guarantees that exactly one agent has to stay in a vertex that correspond to one of the literals x and $\overline{x}$.

For the first direction, assume $\Phi =$ **true**. Consider the strategy F_1 for Coverer that chooses an assignment to the existential variables, depending on the assignments chosen for preceding variables, in a way that ensures φ is satisfied. Then, for every literal vertex that corresponds to universal variables the play encounters, F_1 leaves one of the agents to traverse its self-loop of indefinitely. Thus, the objective α_x is satisfied in the play that stays in the literal vertex x or $\overline{x}$, for every $x \in X$. And, since the chosen assignment satisfies φ, and literal vertices are visited infinitely often iff they are evaluated to **true** in the chosen assignment, every objective α_{C_i} is satisfied in at least one play. Therefore, F_1 is a covering strategy.

For the second direction, assume there exists a covering strategy F_1 for Coverer in $\mathcal{G}$. Note that by the definition of the construction, F_1 allocates a single agent to stay indefinitely in every literal vertex the play reaches. Also note that for every strategy f_2 for Disruptor, the set of literal vertices that the plays visit infinitely often induce an assignment that satisfy φ. Indeed, otherwise there exists a strategy f_2 for Disruptor that induces an assignment for which there exists a clause C_i all whose literals are evaluated to **false**, thus α_{C_i} is not covered in $\mathsf{outcome}(\langle F_1, f_2\rangle)$, contradicting the fact that F_1 is a covering strategy. Thus, there exists an assignment to the existential variables, depending on assignments to preceding universal variables, that guarantee the sanctification of φ. Accordingly, $\Phi =$ **true**.

A.5 Missing details in the proof of Theorem 14

Consider a strategy $F_1 = \langle f_1^1, \ldots, f_1^k \rangle$ for Coverer. For every $i \in [k]$, let $\rho_i = \mathsf{outcome}(\langle f_1^i, f_2' \rangle)$. Since f_2' is a winning strategy for the ExistsC objective $\{\cup(\beta \setminus \delta)\}_{\delta \in \Delta_{f_2}}$, for every $i \in [k]$ there exist $\delta_i \in \Delta_{f_2}$ such that ρ_i satisfies the co-Büchi objective $\cup(\beta \setminus \delta_i)$, and so the set of Büchi objectives that ρ_i satisfies is a subset of δ_i. Since $\delta_1, \ldots, \delta_k \in \Delta_{f_2}$ and f_2 is a disrupting strategy in $\mathcal{G}$, then by Lemma 13, we have that $\bigcup_{i \in [k]} \delta_i \neq \beta$. Thus, there exists an objective $\alpha_j \in \beta$ such that for every $i \in [k]$, α_j is not satisfied in ρ_i. Accordingly, we have that f_2' is a disrupting strategy for Disruptor in $\mathcal{G}$.

A.6 Missing details in the proof of Theorem 20

First, for the NP part, the algorithm checks that there exists a U-trap strategy for Disruptor that respects g. For that, the algorithm checks that Disruptor can force the play to reach U, checks that Coverer cannot force the play to leave U by verifying that for every vertex $v \in V_1 \cap U$, we have that $\mathsf{succ}(v) \subseteq U$, and then checks that a U-trap strategy can respect g by verifying that for every vertex $v \in V_2 \cap U$, we have that $g(v) \subseteq \mathsf{succ}(v)$.

Then, for the co-NP oracle, the algorithm checks that every U-trap strategy for Disruptor that respects g is a disrupting strategy. For that, the algorithm checks that for every U-trap strategy f_2 for Disruptor that respects g and every strategy $F_1 = \langle f_1^1, \ldots, f_1^k \rangle$ for Coverer, we have that β is not covered in $\mathsf{outcome}(\langle F_1, f_2 \rangle)$. That is, $\bigcap_{i \in [k]} \{\alpha_j \in \beta : \mathit{inf}(\mathsf{outcome}(\langle f_1^i, f_2 \rangle)) \cap \alpha_j \neq \emptyset\} \neq \emptyset$. For that, the algorithm checks that for every k sets $U_1, \ldots, U_k \subseteq U$ of vertices such that for every $i \in [k]$ we have that $U_i = \mathit{inf}(\mathsf{outcome}(\langle f_1, f_2 \rangle))$ for some strategy f_1 for an agent of Coverer and a U-trap strategy f_2 for Disruptor that respects g, we have that $\bigcap_{i \in [k]} \{\alpha_j \in \beta : U_i \cap \alpha_j \neq \emptyset\} \neq \emptyset$. Note that for every set of vertices $U \subseteq V$ and a fairness requirement g for Disruptor, for every U-trap strategy f_2 for Disruptor that respects g and a strategy f_1 for an agent of Coverer, we have that $U' = \mathit{inf}(\mathsf{outcome}(\langle f_1, f_2 \rangle))$ is a strongly connected subset of U such that for every vertex $v \in V_2 \cap U'$ we have that $g(v) \subseteq U'$, and the vertices in $U' \setminus \{g(v) : v \in V_2 \cap U'\}$ are visited infinitely often following choices made by f_1, and thus for every vertex $u \in U' \setminus \{g(v) : v \in V_2 \cap U'\}$ we have that $u \in \{\mathsf{succ}(v) : v \in V_1 \cap U'\}$. Thus, the co-NP oracle checks that for a guessed set of k sets $U_1, \ldots, U_k \subseteq U$ of vertices, for every $i \in [k]$, we have that U_i is strongly connected, $\{g(v) : v \in V_2 \cap U_i\} \subseteq U_i$, and $U_i \setminus \{g(v) : v \in V_2 \cap U_i\} \subseteq \{\mathsf{succ}(v) : v \in V_1 \cap U_i\}$.

A.7 Proof of Theorem 21

We describe reductions from 2QBF. Consider a 2QBF formula $\Phi = \exists X \forall Y \varphi$ with $\varphi = C_1 \vee \cdots \vee C_m$ and $C_i = (l_i^1 \wedge l_i^2 \wedge l_i^3)$, for every $i \in [m]$. For both $\gamma \in \{\text{B,C}\}$, we construct from Φ a γ-CG $\mathcal{G}$ such that $\Phi = \mathbf{true}$ iff Disruptor has a disrupting strategy in $\mathcal{G}$.

We define the γ-CG $\mathcal{G}$ as in the proof of Theorem 11, for the QBF formula $\overline{\Phi} = \forall X \exists Y \overline{\varphi}$. Note that since φ is in 3DNF, the formula $\overline{\varphi}$ is in 3CNF. We prove the correctness of the construction. That is, we prove that $\Phi = \mathbf{true}$ iff Disruptor has a disrupting strategy in $\mathcal{G}$.

For the first direction, assume $\Phi = \mathbf{true}$. Let $\xi : X \to \{\mathbf{true}, \mathbf{false}\}$ be an assignment to the variables in X such that for every assignment $\zeta : Y \to \{\mathbf{true}, \mathbf{false}\}$ to the variables in Y, we have that ξ and ζ satisfy φ. Consider a strategy f_2 for Disruptor that chooses an assignment to the variables in X according to ξ. That is, for every variable $x \in X$, the strategy f_2 proceeds to the literal vertex x if $\xi(x) = \mathbf{true}$, and otherwise proceeds to the literal vertex $\overline{x}$. It is not hard to see that f_2 is a disrupting strategy for Disruptor in

$\mathcal{G}$. Indeed, for every strategy F_1 for Coverer that allocates one agent to stay in the literal vertices of each variable, there exists a clause $\overline{C_i}$ of $\overline{\varphi}$ all whose literals are evaluated to **false** in the corresponding assignment, thus the objective $\alpha_{\overline{C_i}}$ is not covered in $\mathsf{outcome}(\langle F_1, f_2\rangle)$.

For the second direction, assume $\Phi = \mathbf{false}$. Then, $\overline{\Phi} = \mathbf{true}$, and so Coverer has a covering strategy in $\mathcal{G}$, as shown in the proof of Theorem 11. Consequentially, Disruptor does not have a disrupting strategy in $\mathcal{G}$.

Arbitrary-Arity Tree Automata for QCTL

François Laroussinie
IRIF, Université Paris Cité, France

Nicolas Markey
IRISA – Inria, CNRS, Univ. Rennes 1, France

Abstract

We introduce a new class of automata (which we coin EU-automata) running on infinite trees of arbitrary (finite) arity. We develop and study several algorithms to perform classical operations (union, intersection, complement, projection, alternation removal) for those automata, and precisely characterise their complexities. We also develop algorithms for solving membership and emptiness for the languages of trees accepted by EU-automata. We then use EU-automata to obtain several algorithmic and expressiveness results for the temporal logics QCTL and QCTL* (which extends CTL and CTL* with quantification over atomic propositions) and for MSO.

2012 ACM Subject Classification Theory of computation → Logic and verification

Keywords and phrases Model-checking, Verification, Automata theory, Quantified CTL

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2025.28

Related Version *Full Version*: `https://arxiv.org/abs/2410.18799` [17]

Acknowledgements We thank Igor Walukiewicz for answering our questions on MSO-automata, and the reviewers for their careful reading and constructive comments.

1 Introduction

Logics and automata. The very tight links between logics and automata on infinite words and trees date back to the early 1960's with the seminal works of Büchi, Elgot, Trakhtenbrot, McNaughton and Rabin [4, 8, 26, 21, 23]. These early results were mainly concerned with the Monadic Second-Order Logic (MSO), and have been further extended to many other logical formalisms such as modal, temporal and fixpoint logics [24, 28, 2, 13, 32]. Those tight links are embodied as translations back and forth between various logical languages and corresponding classes of automata; translations from logics to automata have allowed to derive efficient algorithms for satisfiability or model checking on the one hand [27, 9, 2]; with additional translations from automata to logics, we get effective ways for proving expressiveness or succinctness results for some of those logics [29, 31, 19, 16, 33]. In this paper, we introduce a new flavour of automata running over trees of arbitrary arity [29, 31, 16], develop and analyse tools to manipulate them, and derive algorithmic and expressiveness results for MSO and the temporal logic Quantified CTL (QCTL) [14, 15, 11, 7] and its fragments.

QCTL. QCTL extends the classical temporal logic CTL with quantification on atomic propositions. For instance, formula $\exists p.\phi$, where ϕ is a CTL formula, states that there exists a labelling of the model under scrutiny with proposition p under which ϕ holds. QCTL is (much) more expressive than CTL: as an example, formula $\exists p.\ (\mathbf{EF}(\phi \wedge p) \wedge \mathbf{EF}(\phi \wedge \neg p))$ expresses the fact that there are at least two reachable states where ϕ holds. The extension of CTL with only *existential* quantification was first studied in [10, 14]: contrary to CTL, the resulting logic is sensitive to unwinding and duplication of transitions; the semantics thus depends on whether the extra labelling refers to the Kripke structure under scrutiny, or on its computation tree. Our sample formula above expresses that there are at least two

© François Laroussinie and Nicolas Markey;
licensed under Creative Commons License CC-BY 4.0

different reachable control states satisfying ϕ in the former case (which we call the *structure semantics*), while it only requires that two different paths lead to some ϕ-states (possibly two copies of the same control state) in the latter semantics (called the tree semantics hereafter). In this paper, we will only consider QCTL with the tree semantics. In that setting, it turns out that QCTL is as expressive as MSO over infinite trees [18].

Tree automata. We use (top-down[1]) tree-automata techniques to study QCTL. Several results already exist on this topic [10, 14, 18], but they all rely on fixed-arity tree automata.

The limitation has several drawbacks. When dealing with model checking, it implies that the compilation of the formula being checked into a tree automaton depends on the (size of the) structure under scrutiny. In particular, it cannot be used directly for evaluating the *program complexity* of QCTL model checking, as it requires bounding the size of the structures that the automaton can handle. An indirect solution to this problem is given in [18], by replacing nodes of arbitrary (finite) arity with binary-tree gadgets. A similar problem occurs when dealing with satisfiability: one has to use additional results to ensure that looking for a structure with bounded size is sufficient. More importantly, when deriving expressiveness results, using fixed-arity tree automata again restricts the results to trees or structures with bounded branching.

In order to handle trees of arbitrary branching degree, tree automata must have a *symbolic* way of expressing transitions, with a finite representation that can cope with any arity. Several solutions have been proposed [1, 13, 29, 31, 16]; we highlight two of them:

- Janin and Walukiewicz introduce MSO-automata [13, 29], in which transitions are defined as first-order formulas: quantification is over the successors of the current node, and predicates indicate in which states of the automaton those successors must be explored. These automata are shown to be as expressive as MSO, and several expressiveness results have been obtained from this construction [13, 30, 33]. However, to the best of our knowledge, the exact complexity of the operations for manipulating those automata has not been studied, and no bounds on the size and complexity of the translations can be derived without a more careful study.
- Wilke introduces $\{\Box, \Diamond\}$-automata [31], which are alternating tree automata with $\Box q$ and $\Diamond q$ as basic blocks for expressing transitions: the former requires that all successors be explored in state q, while the latter asks that some successor be explored in state q. Any CTL formula can be turned into an equivalent $\{\Box, \Diamond\}$-automaton of linear size; this is used to prove that the extension CTL^+ of CTL is exponentially more succinct that CTL. However, $\{\Box, \Diamond\}$-automata are not expressive enough to capture MSO or QCTL.

Our contribution. In this paper, we define a new class of symmetric arbitrary-arity alternating tree automata, develop effective operations for their manipulation, and study the complexity of those operations and the size of the resulting automata. Instead of using pairs (k, q) in the transition function to specify that the k-th successor of the current node has to be accepted by the automaton in state q, transitions of our automata are defined with pairs $\langle E; U\rangle$, where E is a multiset of states that have to occur among the set of states involved in the exploration of the successors of the current node, while U is a set of states indicating which states are allowed for exploring successor nodes that are not explored by

[1] There are several families of tree automata: top-down tree automata explore (finite or infinite) trees starting from the root; bottom-up tree automata explore finite trees from the leaves up to the root; tree-walking automata are a kind of two-way automata for trees. We refer to [6, 3] for more details.

states of E. For example, $\langle E = \{\!\{q, q, q'\}\!\}; U = \{q''\}\rangle$ requires the presence of at least three successors nodes; two successors will be explored in state q, one in state q', and the remaining ones (if any) in state q''. We name those automata EU-automata[2].

It is not hard to prove that such automata are closed under conjunction and disjunction, thanks to alternation. Closure under negation is harder to prove: while $\Box$ and $\Diamond$ are dual to each other, which provides an easy complementation procedure for $\{\Box, \Diamond\}$-automata, there is no obvious way of expressing the negation of EU-pairs in terms of EU-pairs. We develop such a translation, and obtain an exponential complementation procedure for EU-automata.

We show that non-alternating EU-automata are also closed under projection, which is the operation we need to encode quantification over atomic propositions of QCTL, and first- and second-order quantification of MSO. Finally we prove that any alternating EU-automaton can be turned into an equivalent non-alternating EU-automaton. For this, we adapt the simulation procedure developed in [29, 33] for MSO-automata to our setting, and evaluate its exact complexity. Note that we use a fine-tuned notion of size for our EU-automata, with different components (number of states, size of Boolean formulas in the transition function, size of EU-pairs, number of (parity) priorities); this allows us to precisely estimate the complexity of these operations.

Putting all the pieces together, we prove that any QCTL formula φ can be turned into an equivalent EU-automaton $\mathcal{A}_\varphi$. The size of the automaton is k-exponential in the size of φ, where k is the number of nested quantifier blocks in φ. This construction then yields optimal algorithms for model-checking and satisfiability for QCTL. Conversely, we prove that acceptance by any EU-automaton can be expressed as a simple QCTL formula. We obtain similar results for QCTL* and MSO. Therefore EU-automata, QCTL, and MSO all characterise exactly the same tree languages.

2 Definitions

2.1 Preliminary definitions

Sets and multisets. Let $\mathcal{S}$ be a finite set. A *multiset* over $\mathcal{S}$ is a mapping $\mu\colon \mathcal{S} \to \mathbb{N}$. Sets are seen as special cases of multisets taking values in $\{0, 1\}$. We use double-brace notation to distinguish between sets and multisets: $\{a, a, a\}$ is the same as the set $\{a\}$ with one element, while $\{\!\{a, a, a\}\!\}$ is the three-element multiset $a \mapsto 3$. The *empty multiset* is the multiset mapping all elements of $\mathcal{S}$ to zero; we denote it with $\varnothing$.

The *support of a multiset* μ is the set $\mathsf{supp}(\mu) = \{s \in \mathcal{S} \mid \mu(s) > 0\}$. The *size* $|\mu|$ of μ is the sum $\sum_{s\in\mathcal{S}} \mu(s)$. For two multisets μ and μ', we write $\mu \sqsubseteq \mu'$, and say that μ is a submultiset of μ', whenever $\mu(s) \leq \mu'(s)$ for all $s \in \mathcal{S}$. This defines a partial ordering over multisets. We write $\mu \sqsubset \mu'$ when $\mu \sqsubseteq \mu'$ and $\mu \neq \mu'$. We define the following operations on multisets: $\mu \uplus \mu'\colon s \in \mathcal{S} \mapsto \mu(s) + \mu'(s)$ and $\mu' \setminus \mu\colon s \in \mathcal{S} \mapsto \max(0, \mu'(s) - \mu(s))$.

Fix a second set $\mathcal{S}'$. For any $c = (s, s') \in \mathcal{S} \times \mathcal{S}'$, we define $\mathsf{proj}_1(c) = s$ and $\mathsf{proj}_2(c) = s'$.

Markings. Let $\mathcal{S}$ and $\mathcal{S}'$ be two finite sets. A *marking* of $\mathcal{S}'$ by $\mathcal{S}$ is a mapping $\nu\colon \mathcal{S}' \to 2^{\mathcal{S}} \setminus \{\varnothing\}$ decorating each element of $\mathcal{S}'$ with a (non-empty) subset of $\mathcal{S}$. A marking ν is a *submarking* of a marking ν', denoted $\nu \sqsubseteq \nu'$, whenever $\nu(s') \subseteq \nu'(s')$ for all $s' \in \mathcal{S}'$.

[2] In [16], Kupferman and Vardi define another variant of arbitrary-arity alternating tree automata in which transitions are based on pairs (U, E). Those automata are equivalent to Wilke's $\{\Box, \Diamond\}$-automata.

A marking ν is *unitary* when $|\nu(s')| = 1$ for all $s' \in \mathcal{S}'$; unitary markings can be seen as mappings from $\mathcal{S}'$ to $\mathcal{S}$. For a unitary marking ν and a subset T of $\mathcal{S}'$, we write $\nu(T)$ for the multiset μ over $\mathcal{S}$ defined as $\mu(s) = \#\{t \in T \mid \nu(t) = s\}$, which we may also write as $\{\!\{\nu(t) \mid t \in T\}\!\}$. We write $\mathsf{img}(\nu)$ for the multiset $\nu(\mathcal{S}')$.

Words and trees. Let Σ be a finite set. A *word* over Σ (or Σ-*word*) is a sequence $w = (w_i)_{0 \le i < k}$ of elements of Σ, with $k \in \mathbb{N} \cup \{+\infty\}$. The *length* (or *size*) of w, denoted with $|w|$, is k. We write Σ^* for the set of finite words over Σ, and Σ^∞ for the set of infinite words over Σ. We write ε for the *empty word* (the only word of size 0).

Let $\mathcal{D}$ be a finite set. A *tree structure* over $\mathcal{D}$ (or $\mathcal{D}$-*tree*) is a subset $t \subseteq \mathcal{D}^*$ that is closed under prefix. The empty word ε is its *root*. The elements of a tree are called nodes. A node m in t is a *successor* of a node n if $m = n \cdot d$ for some $d \in \mathcal{D}$. In that case, n is the (unique) predecessor of m. We write $\mathsf{succ}(n)$ for the set of successors of node n. Notice that in a $\mathcal{D}$-tree, any node may have at most $|\mathcal{D}|$ successors; this integer $|\mathcal{D}|$ is the *arity* of the tree. A *branch* of a tree is a (finite of infinite) sequence $b = (n_i)_{0 \le i < k}$ of nodes of the tree such that $n_0 = \varepsilon$, n_{i+1} is a successor of n_i for all $0 \le i < k - 1$, and if k is finite, n_{k-1} is a leaf. The value of $k \in \mathbb{N} \cup \{+\infty\}$ is the length of b, denoted with $|b|$.

A Σ-*labelled* $\mathcal{D}$-*tree* is a pair $\mathcal{T} = (t, l)$ where t is a $\mathcal{D}$-tree and $l \colon t \to \Sigma$ labels each node of t with a letter in Σ. With any branch $b = (n_i)_{0 \le i < k}$ of t in a Σ-labelled $\mathcal{D}$-tree $\mathcal{T} = (t, l)$, we associate its *word* $w(b)$ over Σ as the word $(w_i)_{0 \le i < k}$ defined as $w_i = l(n_i)$ for all $0 \le i < k$.

2.2 Automata over trees of arbitrary arity

In this section, we introduce our automata running over trees of arbitrary arity. The core element of their transition functions are EU-pairs and EU-constraints.

EU-pairs and EU-constraints. Let $\mathcal{S}$ be a countable set. An *EU-pair* over $\mathcal{S}$ is a pair $\langle E; U\rangle \in \mathbb{N}^{\mathcal{S}} \times 2^{\mathcal{S}}$, where E is a multiset over $\mathcal{S}$ and U is a subset of $\mathcal{S}$. A multiset μ over $\mathcal{S}$ satisfies the EU-pair $\langle E; U\rangle$, denoted $\mu \Vdash \langle E; U\rangle$, whenever $E \sqsubseteq \mu$ and $\mathsf{supp}(\mu \setminus E) \subseteq U$. We write $\mathsf{EU}(\mathcal{S}) = \mathbb{N}^{\mathcal{S}} \times 2^{\mathcal{S}}$ for the set of EU-pairs over $\mathcal{S}$.

▶ **Example 1.** Consider a set $\mathcal{S} = \{q_1, q_2, q_3, q_4\}$. The EU-pair $\langle q_1 \mapsto 3, q_2 \mapsto 1; \{q_1, q_3\}\rangle$ characterises all multisets containing *at least* three occurrences of q_1, *exactly* one occurrence of q_2, an arbitrary number of occurrences of q_3, and no occurrences of q_4. ⌟

For a finite set $\mathcal{B}$ of variables, we write $\mathsf{PBF}(\mathcal{B})$ for the set of *positive boolean combinations* over $\mathcal{B}$: $\mathsf{PBF}(\mathcal{B}) \ni \phi ::= \top \mid \bot \mid v \mid \phi \wedge \phi \mid \phi \vee \phi$ where v ranges over $\mathcal{B}$. The set of *disjunctions* over $\mathcal{B}$, denoted $\mathsf{DBF}(\mathcal{B})$, is the subset of $\mathsf{PBF}(\mathcal{B})$ where conjunctions are not allowed.

An *EU-constraint* is a positive boolean formula over EU-pairs (that is an element of $\mathsf{PBF}(\mathsf{EU}(\mathcal{S}))$) whose semantics is defined as follows:

▶ **Definition 2.** *Let $\mathcal{S}$ and $\mathcal{S}'$ be two countable sets. Let $\langle E; U\rangle$ be an EU-pair over $\mathcal{S}$, and ν be a marking of $\mathcal{S}'$ by $\mathcal{S}$. Then ν satisfies $\langle E; U\rangle$, denoted $\nu \models\!\!\models \langle E; U\rangle$, if there exists a unitary marking $\nu' \sqsubseteq \nu$, such that $\mathsf{img}(\nu') \Vdash \langle E; U\rangle$.*

This definition extends to EU-constraints inductively as follows:

- *$\nu \models\!\!\models \phi_1 \vee \phi_2$ if, and only if, $\nu \models\!\!\models \phi_1$ or $\nu \models\!\!\models \phi_2$;*
- *$\nu \models\!\!\models \phi_1 \wedge \phi_2$ if, and only if, $\nu \models\!\!\models \phi_1$ and $\nu \models\!\!\models \phi_2$.*

Notice that $\nu \models\!\!\models \phi$ is *not* equivalent to having an unitary marking $\nu' \sqsubseteq \nu$ satisfy ϕ, since different submarkings ν' may be needed for different EU-pairs. However, the equivalence holds if ϕ is a disjunction of EU-pairs, since in that case a single EU-pair has to be fulfilled.

EU tree automata. We can now define our class of automata:

▶ **Definition 3.** *Let Σ be a finite alphabet. An* alternating EU parity tree automaton *(*AEUPTA *for short) over Σ is a 4-tuple $\mathcal{A} = (Q, q_{init}, \delta, \omega)$ with*

- *Q is a finite set of states, and $q_{init} \in Q$ is the initial state;*
- *$\delta \colon Q \times \Sigma \to \mathit{PBF}(\mathit{EU}(Q))$ is the set of transitions;*
- *$\omega \colon Q \to \mathbb{N}$ is a* priority *function defining the acceptance condition (as a* parity condition*).*

An AEUPTA is non-alternating[3] *(and is thus an* EU parity tree automaton*,* EUPTA *for short) if δ takes values in $\mathit{DBF}(\mathit{EU}(Q))$.*

We can then define the notion of execution tree of an AEUPTA:

▶ **Definition 4.** *Let $\mathcal{A} = (Q, q_{init}, \delta, \omega)$ be an AEUPTA over Σ and $\mathcal{T} = (t, l)$ be a Σ-labelled $\mathcal{D}$-tree, for some finite set $\mathcal{D}$. An* execution tree *of $\mathcal{A}$ over $\mathcal{T}$ is a $(t \times Q)$-labelled $(\mathcal{D} \times Q)$-tree $\mathcal{U} = (u, \ell)$ such that*

- *the root ε_u of u is labelled with ε_t and q_{init} (formally, $\ell(\varepsilon_u) = (\varepsilon_t, q_{init})$);*
- *any non-root node $n_u = (d_i, q_i)_{0 \le i < |n_u|}$ of u is labelled with $\ell(n_u) = ((d_i)_{0 \le i < |n_u|}, q_{|n_u|-1})$;*
- *for any node n_u of the form $(d_i, q_i)_{0 \le i < |n_u|}$ of u with $\ell(n_u) = (m_t, q)$, letting ν_{n_u} be the marking of $\mathsf{succ}(m_t)$ by Q such that $\nu_{n_u}(m_t \cdot d) = \{q' \in Q \mid n_u \cdot (d, q') \in \mathsf{succ}(n_u)\}$, we have $\nu_{n_u} \models \delta(q, l(m_t))$. We name this marking ν_{n_u} the marking of $\mathsf{succ}(m_t)$* induced *by $\mathcal{U}$.*

The tree $\mathcal{T}$ is accepted *by $\mathcal{A}$ if there exists an execution tree $\mathcal{U}$ of $\mathcal{A}$ over $\mathcal{T}$ such that any infinite branch b is* accepting*, i.e., the least priority $\omega_{\min}(b)$ appearing infinitely many times along b is even. Such an execution tree is said to be* accepting*. The* language *of $\mathcal{A}$, denoted by $\mathcal{L}(\mathcal{A})$, is the set of all trees accepted by $\mathcal{A}$.*

Clearly, if a marking ν satisfies an EU-constraint, then so does any marking ν' containing ν. that $\mathcal{D}$ is not constrained by the definition of AEUPTAs, so that AEUPTAs may accept trees of arbitrary (finite) arity. However, the multisets in the "existential part" of EU-pairs can be used to impose a lower bound on the number of successors for the EU-pair to be satisfied, and an upper bound can be imposed by letting the universal part be empty.

▶ **Example 5.** Consider a node m of some input tree $\mathcal{T}$, with three successors $m \cdot d_1$, $m \cdot d_2$ and $m \cdot d_3$; assume that this node m is labelled with some letter σ. Consider an AEUPTA $\mathcal{A}$ visiting node m in state q, giving rise to a node n in an execution tree $\mathcal{U}$ with $\ell(n) = (m, q)$. The successors of n in the execution tree give rise to the marking ν_n such that $\nu_n(m \cdot d_1) = \{q_1, q_3\}$, $\nu_n(m \cdot d_2) = \{q_2, q_4\}$, and $\nu_n(m \cdot d_3) = \{q_1, q_4\}$, as depicted in Fig. 1.

Assume that $\delta(q, \sigma)$ is satisfied by the following set W of EU-pairs:

$$W = \Big\{ \underbrace{\langle q_1 \mapsto 2; \{q_2\}\rangle}_{\langle E_1; U_1\rangle},\ \underbrace{\langle q_1 \mapsto 1, q_2 \mapsto 1, q_3 \mapsto 1; \{q_4\}\rangle}_{\langle E_2; U_2\rangle},\ \underbrace{\langle q_3 \mapsto 1; \{q_4\}\rangle}_{\langle E_3; U_3\rangle} \Big\}.$$

Figure 1 displays a possible set of successors $\mathsf{succ}(n)$ of n in the execution tree $\mathcal{U}$. Using the following three submarkings ν_i of ν_n, we are able to fulfill all three EU-pairs of W: $\nu_1 \colon n \cdot d_1 \mapsto q_1, n \cdot d_2 \mapsto q_2, n \cdot d_3 \mapsto q_1$, $\nu_2 \colon n \cdot d_1 \mapsto q_3, n \cdot d_2 \mapsto q_2, n \cdot d_3 \mapsto q_1$, and $\nu_3 \colon n \cdot d_1 \mapsto q_3, n \cdot d_2 \mapsto q_4, n \cdot d_3 \mapsto q_4$. ⌟

[3] Such automata are usually said to be *non-deterministic* in the litterature. We prefer to name them *non-alternating* since they are the class of automata we get with our alternation-removal procedure, and also because we do not have a notion of being deterministic for our tree automata (an EUPTA has in general more than one execution trees on an input tree).

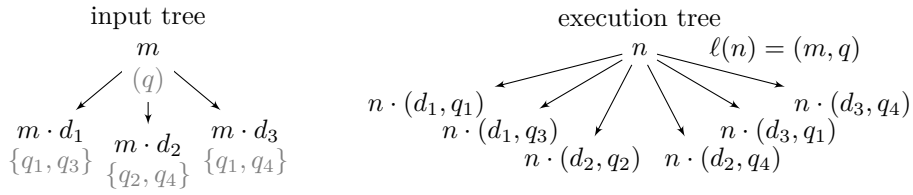


Figure 1 Example of a transition of the automaton when exploring node m in some state q.

▶ **Example 6.** To illustrate the use of AEUPTA, we display an example of an automaton accepting all trees satisfying the following two conditions: exactly two infinite branches contain infinitely many occurrences of a, and at least one branch is fully labelled with b.

Let $\Sigma = \{a, b, c\}$. We let $\mathcal{A} = (Q, q_{init}, \delta, \omega)$, where the set Q of states is $\{q_{init}, q_a^{2i}, q_a^{1i}, r_a^i, q_b^\infty, q_a^f, r_a^f, q_\top\}$: the automaton uses state q_a^{2i} to visit a prefix of a branch having exactly two infinite sub-branches where a occurs infinitely many times; it uses states q_a^{1i} and r_a^i to visit subtrees in which exactly one branch has infinitely many occurrences of a: state r_a^i is used when letter a is read, and the acceptance condition requires that it be visited infinitely many times along some branch; similarly, states q_a^f and r_a^f are used to explore subtrees in which all branches have finitely many occurrences of a. Finally, we use q_b^∞ to explore branches fully labelled with b. Following this intuition, the transition function is defined as:

$$\delta(q_{init}, \sigma) = \begin{cases} \Big(\langle q_a^{2i} \mapsto 1; \{q_a^f\}\rangle \vee \langle q_a^{1i} \mapsto 2; \{q_a^f\}\rangle\Big) \wedge \langle q_b^\infty \mapsto 1; \{q_\top\}\rangle & \text{if } \sigma = b \\ \bot & \text{otherwise} \end{cases}$$

$$\delta(q_a^{2i}, \sigma) = \langle q_a^{2i} \mapsto 1; \{q_a^f\}\rangle \vee \langle q_a^{1i} \mapsto 2; \{q_a^f\}\rangle \quad \text{for any } \sigma \in \Sigma$$

$$\delta(q_a^{1i}, \sigma) = \delta(r_a^i, \sigma) = \begin{cases} \langle r_a^i \mapsto 1; \{q_a^f\}\rangle & \text{if } \sigma = a \\ \langle q_a^{1i} \mapsto 1; \{q_a^f\}\rangle & \text{otherwise} \end{cases}$$

$$\delta(q_a^f, \sigma) = \delta(r_a^f, \sigma) = \begin{cases} \langle \varnothing; \{r_a^f\}\rangle & \text{if } \sigma = a \\ \langle \varnothing; \{q_a^f\}\rangle & \text{otherwise} \end{cases}$$

$$\delta(q_b^\infty, \sigma) = \begin{cases} \langle q_b^\infty \mapsto 1; \{q_\top\}\rangle & \text{if } \sigma = b \\ \bot & \text{otherwise} \end{cases}$$

Finally, $\delta(q_\top, \sigma) = \top$ for any $\sigma \in \Sigma$, so that any subtree is accepted when explored in $q_\top$. Notice that $\delta(q_{init}, \sigma)$ contains a conjunction, so that the automaton is alternating. Notice also how the disjunctive EU-constraint in $\delta(q_{init}, \sigma)$ and $\delta(q_a^{2i}, \sigma)$ will either look for a single successor from which two branches will have infinitely many occurrences of a, or for two successors from each of which there will be such a branch. All other branches will be checked to have finitely many occurrences of a by exploring them in state q_a^f. The priority function is defined so as to check that a occurs infinitely many times along branches where it has to:

$$\omega(r_a^i) = \omega(q_b^\infty) = 0 \qquad \omega(q_a^{1i}) = \omega(q_a^{2i}) = \omega(r_a^f) = 1 \qquad \omega(q_a^f) = 2$$

Figures 2 and 3 display (part of) an input tree and a corresponding execution tree for the automaton built above The root of the execution tree is labelled with (ε, q_{init}) and the marking ν induced by the execution tree for the successors of the root of the input tree is $\{d_1 \mapsto \{q_\top, q_a^{1i}\}, d_2 \mapsto \{q_a^{1i}, q_b^\infty\}, d_3 \mapsto \{q_\top, q_a^f\}\}$, which satisfies $\delta(q_{init}, b)$: indeed, the unitary marking $\{d_1 \mapsto \{q_a^{1i}\}, d_2 \mapsto \{q_a^{1i}\}, d_3 \mapsto \{q_a^f\}\}$ fulfills $\langle q_a^{1i} \mapsto 2; \{q_a^f\}\rangle$, and the unitary marking $\{d_1 \mapsto \{q_\top\}, d_2 \mapsto \{q_b^\infty\}, d_3 \mapsto \{q_\top\}\}$ fulfills $\langle q_b^\infty \mapsto 1; \{q_\top\}\rangle$. ⌟

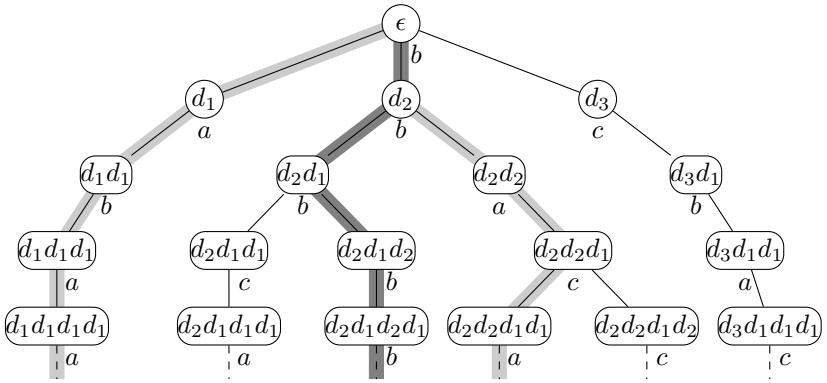


Figure 2 An input tree with exactly two branches (light grey) having infinitely many occurrecnes of a and at least one branch (dark grey) fully labelled with b.

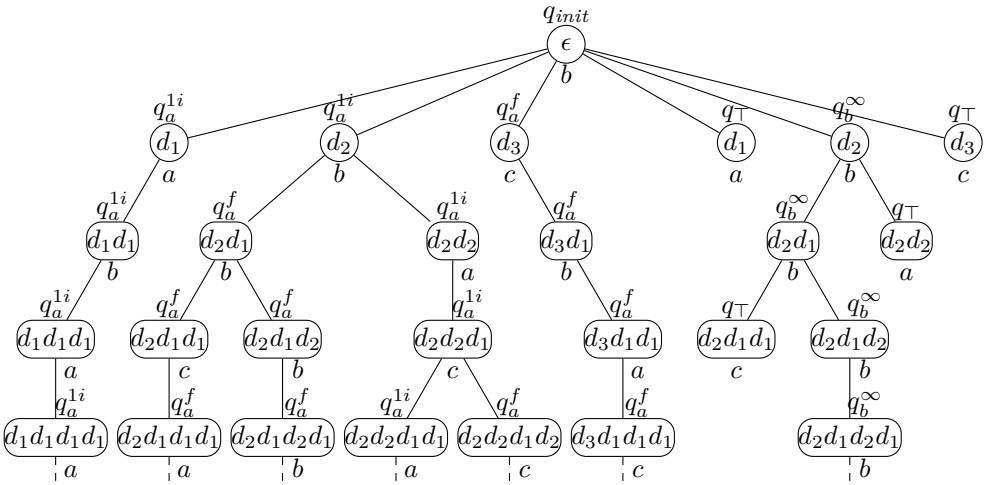


Figure 3 An execution tree for our automaton $\mathcal{A}$ of Example 6 on the input tree of Fig. 2 (notice that, for the sake of readability, we keep the names of the nodes as in the input tree, using words over $\mathcal{D}$ instead of words over $\mathcal{D} \times Q$). Because of the conjunction in $\delta(q_{init}, b)$, the automaton explores the input tree twice, so that the execution tree contains two copies of the input tree: the subtree to the left of the root corresponds to the part looking for two branches with infinitely many occurrences of a, while the subtree to the right looks for a branch fully labelled with b.

Size of an AEUPTA. The *size of an AEUPTA*, denoted with $|\mathcal{A}|$, is a 5-tuple $(|Q|, |\delta|_{\mathsf{Bool}}, |\delta|_{\mathsf{E}}, |\delta|_{\mathsf{U}}, |\omega|)$, where $|\delta|_{\mathsf{Bool}}$ is the maximum size of the boolean formulas in δ, $|\delta|_{\mathsf{E}}$ (resp. $|\delta|_{\mathsf{U}}$) is the size of the largest existential part E (resp. universal part U) in some EU-pair in δ, and $|\omega| = |\{\omega(q) \mid q \in Q\}|$ is the number of priorities used in the automaton. Note that contrary to classical, fixed-arity tree automata, we explicitly consider the size of the transition function δ in the size of an AEUPTA; this is motivated by the fact that EU-constraints may succinctly encode very complex transitions, regardless of the size of Q.

In the sequel, we say that the size of an AEUPTA is at most $(s_Q, s_B, s_E, s_U, s_\omega)$ when $|Q| \leq s_Q$, $|\delta|_{\mathsf{Bool}} \leq s_B$, $|\delta|_{\mathsf{E}} \leq s_E$, $|\delta|_{\mathsf{U}} \leq s_U$ and $|\omega| \leq s_\omega$. The fact that we use five different parameters in the size of AEUPTAs will allow us to have more precise bounds on the complexities of our operations for manipulating them.

▶ Remark 7. It is possible to modify any EU-automaton in such a way that every EU-pair it involves uses only a singleton or the empty set as its universal part (*i.e.*, with $|\delta|_{\mathsf{U}} \leq 1$). This can be performed by replacing any general EU-pair $\langle E; U\rangle$ with $\langle E; \{q_U\}\rangle$, where q_U is a fresh state s.t. $\delta(q_U, \sigma) = \bigvee_{q\in U} \delta(q, \sigma)$, and keeping track of the state being chosen in order to encode the acceptance condition.

This transformation produces an automaton whose size remains polynomial in the size of the original automaton. Moreover, as we will see in the sequel, all the constructions we develop for manipulating AEUPTAs preserve this property of having only singletons or empty sets as the universal part of EU-pairs. ◀

Game-based semantics. Some of our results rely on the tight links between automata and games, which we can extend to our class of AEUPTA. More precisely, given a Σ-labelled $\mathcal{D}$-tree $\mathcal{T}$ and an AEUPTA $\mathcal{A}$, we can build a two-player turn-based parity game $\mathcal{G}_{\mathcal{A},\mathcal{T}}$ enjoying the following property:

▶ **Proposition 8.** *The Σ-labelled $\mathcal{D}$-tree $\mathcal{T}$ is accepted by the AEUPTA $\mathcal{A}$ if, and only if, Player 0 has a winning strategy from the initial state $(\varepsilon_t, q_{init})$ in the parity game $\mathcal{G}_{\mathcal{A},\mathcal{T}}$.*

When $\mathcal{T}$ is regular and corresponds to the execution tree of some finite Kripke structure $\mathcal{K} = (V, E, \ell)$, the construction above can be adapted to give rise to a *finite* game $\mathcal{G}_{\mathcal{A},\mathcal{K}}$ with the same property. The sizes of the state space of $\mathcal{G}_{\mathcal{A},\mathcal{K}}$ and of its transition relation both are in $O(|V| \cdot (|Q| \cdot |\delta|_{\mathsf{Bool}} + |Q|^{|V|}))$. The term $|Q|^{|V|}$ comes from the fact we have to consider all possible unitary markings of the successors of the nodes in $\mathcal{K}$ by Q. It is worth noticing that for AEUPTA involving only simple constraints of the form $\langle q \mapsto 1; \{q_\top\}\rangle$ (where from $q_\top$ any tree is accepted) or $\langle \varnothing; \{q\}\rangle$ those sizes are in $O(|V| \cdot |Q| \cdot (|\delta|_{\mathsf{Bool}} + |V|))$. Such automata actually correspond to $\{\Box, \Diamond\}$-automata.

3 Operations on AEUTAs

This section is the main technical part of our paper: we develop algorithms for performing various operations on AEUPTAs (namely union and intersection, projection, complementation and alternation removal), and carefully study the size of the AEUPTAs we obtain. We will see that there is no surprise with union, intersection and projection: the size of the resulting automata is linear in the sizes of the inputs.

However, projection assumes that the input automaton is non-alternating; hence projection will be used in combination with alternation removal, and this combination involves an exponential blow-up. Thanks to our careful computation of the various parameters of the size of automata (see Table 1), we notice that while complement and alternation removal both involve an exponential blow-up, their combination only yields a single-exponential blow-up. This will allow us to obtain optimal complexity in our applications to QCTL and MSO.

3.1 Union, intersection

Union and intersection are straightforward for AEUPTAs, thanks to alternation.

▶ **Theorem 9.** *Let $\mathcal{A} = (Q, q_{init}, \delta, \omega)$ and $\mathcal{A}' = (Q', q'_{init}, \delta', \omega')$ be two AEUPTAs. There exist AEUPTAs $\mathcal{A}_\cup$ and $\mathcal{A}_\cap$, respectively accepting the union and the intersection of the language of $\mathcal{A}$ and $\mathcal{A}'$, and having size at most $(|Q| + |Q'| + 1, |\delta|_{Bool} + |\delta'|_{Bool} + 1, \max(|\delta|_E, |\delta'|_E), \max(|\delta|_U, |\delta'|_U), \max(|\omega|, |\omega'|) + 1)$. If the minimum priorities of ω and ω' are equal, then the number of priorities in $\mathcal{A}_\cup$ and $\mathcal{A}_\cap$ can be bounded by $\max(|\omega|, |\omega'|)$.*

Table 1 Summary of the size of the automata produced by our algorithms.

	projection	complement	alternation removal
$\lvert Q_{res}\rvert$	$\leq \lvert Q\rvert$	$\in O(\lvert Q\rvert \cdot \lvert\delta\rvert_{\mathsf{Bool}} \cdot \lvert\Sigma\rvert \cdot \lvert\delta\rvert_{\mathsf{E}} \cdot 2^{\lvert\delta\rvert_{\mathsf{E}}})$	$\in 2^{O(\lvert Q\rvert^2 \cdot \log(\lvert Q\rvert))}$
$\lvert\delta_{res}\rvert_{\mathsf{Bool}}$	$\leq \lvert\Sigma'\rvert \cdot \lvert\delta\rvert_{\mathsf{Bool}}$	$\in O(\lvert Q\rvert \cdot \lvert\delta\rvert_{\mathsf{Bool}} \cdot \lvert\delta\rvert_{\mathsf{E}}^3 \cdot 4^{\lvert\delta\rvert_{\mathsf{E}}})$	$\in (2\lvert\delta\rvert_{\mathsf{E}} \cdot \lvert\delta\rvert_{\mathsf{U}})^{O(\lvert Q\rvert^2 \cdot \lvert\delta\rvert_{\mathsf{Bool}}^2 \cdot \lvert\delta\rvert_{\mathsf{E}})}$
$\lvert\delta_{res}\rvert_{\mathsf{E}}$	$\leq \lvert\delta\rvert_{\mathsf{E}}$	$\leq \lvert\delta\rvert_{\mathsf{E}} + 1$	$\leq \lvert Q\rvert \cdot \lvert\delta\rvert_{\mathsf{Bool}} \cdot \lvert\delta\rvert_{\mathsf{E}}$
$\lvert\delta_{res}\rvert_{\mathsf{U}}$	$\leq \lvert\delta\rvert_{\mathsf{U}}$	$\leq \max(\lvert\delta\rvert_{\mathsf{U}}, 1)$	$\leq \lvert\delta\rvert_{\mathsf{U}}^{\lvert Q\rvert \cdot \lvert\delta\rvert_{\mathsf{Bool}}}$
$\lvert\omega_{res}\rvert$	$\leq \lvert\omega\rvert$	$\leq \lvert\omega\rvert + 1$	$\leq 2(\lvert Q\rvert \cdot \lvert\omega\rvert + 1)$

3.2 Projection

Given an AEUPTA $\mathcal{A}$ over alphabet $\Sigma_1 \times \Sigma_2$, *projection* consists in building another AEUPTA $\mathcal{A}_1$, over alphabet Σ_1, accepting all Σ_1-labelled trees whose labelling can be extended on $\Sigma_1 \times \Sigma_2$ to make the tree accepted by $\mathcal{A}$. This is a classical construction, and it can be performed easily on *non-alternating* automata [22].

Formally, two $\Sigma_1 \times \Sigma_2$-labelled trees $\mathcal{T} = (t, l)$ and $\mathcal{T}' = (t', l')$ are said Σ_1-*equivalent*, denoted $\mathcal{T} \equiv_{\Sigma_1} \mathcal{T}'$, whenever $t = t'$ and for any node n of these trees, it holds $\mathsf{proj}_1(l(n)) = \mathsf{proj}_1(l'(n))$. Σ_2-equivalence is defined analogously.

▶ **Theorem 10.** *Let $\mathcal{A} = (Q, q_{init}, \delta, \omega)$ be an EUPTA over $\Sigma = \Sigma_1 \times \Sigma_2$. For each $i \in \{1, 2\}$, we can build an EUPTA $\mathcal{A}_i$ over Σ such that, for any Σ-labelled tree $\mathcal{T}$, it holds: $\mathcal{T} \in \mathcal{L}(\mathcal{A}_i)$ if, and only if, there is a Σ-labelled tree $\mathcal{T}'$ in $\mathcal{L}(\mathcal{A})$ such that $\mathcal{T} \equiv_{\Sigma_i} \mathcal{T}'$. The size of $\mathcal{A}_i$ is at most $(\lvert Q\rvert, \lvert\Sigma_{3-i}\rvert \cdot \lvert\delta\rvert_{Bool}, \lvert\delta\rvert_E, \lvert\delta\rvert_U, \lvert\omega\rvert)$.*

3.3 Complementation

Complementation is usually easy for alternating parity automata: it suffices to dualise the transition function (swapping disjunctions and conjunctions) and shifting the priorities. For our AEUTA however, we have to express the negation of any EU-pair $\langle E; U\rangle$ as an EU-constraint.

The question then is to characterise those nodes that *fail to satisfy* an EU-pair $\langle E; U\rangle$. There can be two reasons for this:

1. either we cannot find $|E|$ successors of the current node n to associate with the $|E|$ states of the existential part,
2. or for every way to satisfy E with nodes in $\mathsf{succ}(n)$, there remain successors that are accepted by no states in U. Equivalently, there is no way to satisfy the existential part E with (at least) all nodes that are accepted by no states in the universal part U. This includes as a special case the situations where we have more than $|E|$ successors that are accepted by no states in U.

Failing to satisfy the existential part of $\langle E; U\rangle$. We first address the case $\langle E; \{q_\top\}\rangle$. When such an EU-pair cannot be satisfied from a tree node x, there exists some pair (F, g) (called a blocking pair hereafter) where $F \sqsubset E$ is a submultiset and $g \in E \setminus F$, such that the constraint $\langle F; \{q_\top\}\rangle$ is satisfied from x and the constraint $\langle F \uplus \{g\}; \{q_\top\}\rangle$ is not satisfied from x. More precisely, one can prove (see Prop. 11, 13 and 14 in [17]) that there exists a blocking pair (F, g) s.t. for any node $y \in \mathsf{succ}(x)$ that is not involved in the satisfaction of F, the subtree rooted at y does not belong to $\mathcal{L}(\mathcal{A}_{g'})$ for any $g' \in \mathsf{supp}(F) \cup \{g\}$, where $\mathcal{A}_{g'}$ denotes the automaton obtained from $\mathcal{A}$ by taking g' as the initial state. Applying this result, the negation of $\langle E; \{q_\top\}\rangle$ can then be expressed as follows:

$$\Phi_E = \bigvee_{F \sqsubset E} \bigvee_{g \in E \setminus F} \langle F; \{(\overline{\mathsf{supp}(F)} \cup \{\overline{g}\}, \wedge)\}\rangle \tag{1}$$

where $(\overline{\mathsf{supp}(F)} \cup \{\overline{g}\}, \wedge)$ is a (single) new state of our automaton from which a subtree will be accepted if, and only if, it does not belong to any $\mathcal{L}(\mathcal{A}_{g'})$ with $g' \in \mathsf{supp}(F) \cup \{g\}$. Notice that for the special case where E is empty, we end up with an empty disjunction, which is equivalent to false (indeed $\langle \varnothing; \{q_\top\}\rangle$ is satisfied by any tree).

Failing to satisfy $\langle E; U\rangle$. We can now address the general case. If an EU-pair $\langle E; U\rangle$ is not satisfied by a node x, then either it has at least $|E|+1$ successors y that are recognized by no automata states $u \in U$, or for some $0 \leq k \leq |E|$, it has at least k successors recognized by no states $u \in U$ and it does not have $|E|$ successors witnessing the satisfaction of $\langle E; \{q_\top\}\rangle$ and of which k successors are accepted by no states $u \in U$ (see Prop. 15 in [17]). This explains the definition of $\overline{\langle E; U\rangle}$ below.

Construction of the complement automaton. We now define the complement automaton $\mathcal{A}^c$ of $\mathcal{A} = (Q, q_{init}, \delta, \omega)$; we let $\mathcal{A}^c = (Q^c, q^c_{init}, \delta^c, \omega^c)$ be such that:

- Q^c is a subset of $D \cup (2^D \times \{\wedge\}) \cup (2^{(2^D \times \{\vee\})} \times \{\wedge\})$, where $D = Q \cup \overline{Q}$, and $\overline{Q} = \{\overline{q} \mid q \in Q\}$ is a set of fresh states. This defines an operator $\bar{\cdot}\colon q \mapsto \overline{q}$ over Q, which we extend to states $\overline{q}$ of $\overline{Q}$ by letting $\overline{\overline{q}} = q$, to subsets X of D by letting $\overline{X} = \{\overline{x} \mid x \in X\}$, to states $(P, \wedge)$ of $2^D \times \{\wedge\}$ by letting $\overline{(P, \wedge)} = (\overline{P}, \vee)$. We finally extend it to $\mathsf{PBF}(Q^c)$ by letting $\overline{\top} = \bot$, $\overline{\bot} = \top$, $\overline{\psi \wedge \phi} = \overline{\psi} \vee \overline{\phi}$ and $\overline{\psi \vee \phi} = \overline{\psi} \wedge \overline{\phi}$.
  Intuitively, from a state $q \in Q$, automaton $\mathcal{A}^c$ will accept the same language as from the same state in $\mathcal{A}$, while from a state $\overline{q} \in \overline{Q}$, it will accept its complement. The language accepted by $\mathcal{A}$ from a state $(P, \wedge) \in 2^D \times \{\wedge\}$ will be the intersection of all languages accepted by $\mathcal{A}$ from all states in P, whereas the language accepted from states of the form $(\{(P_i, \vee) \mid 1 \leq i \leq k\}, \wedge) \in 2^{(2^D \times \{\vee\})} \times \{\wedge\}$ will be the intersection (over i) of the unions of the languages accepted from all states in P_i. Notice that the indications of $\wedge$ and $\vee$ are mainly used for the sake of clarity.
- accordingly, $q^c_{init} = \overline{q_{init}}$;
- δ^c is defined as follows, for $q \in Q$, $P \in 2^D \cup 2^{2^D}$, and $\sigma \in \Sigma$: we let

$$\delta^c(q, \sigma) = \delta(q, \sigma) \qquad \delta^c((P, \wedge), \sigma) = \bigwedge_{r \in P} \delta^c(r, \sigma)$$

$$\delta^c(\overline{q}, \sigma) = \overline{\delta(q, \sigma)} \qquad \delta^c((P, \vee), \sigma) = \bigvee_{r \in P} \delta^c(r, \sigma)$$

  and, following the developments above:

$$\overline{\langle E; U\rangle} = \langle (\overline{U}, \wedge) \mapsto |E| + 1; \{q_\top\}\rangle \vee \bigvee_{0 \leq k \leq |E|} \Big(\langle (\overline{U}, \wedge) \mapsto k; \{q_\top\}\rangle \wedge \bigwedge_{\substack{m \sqsubseteq E \\ |m| = k}} \Phi_{(E \setminus m) \uplus m^{\overline{U}}} \Big)$$

  where $m^{\overline{U}}$ is the multiset defined by $\{(\{x\} \cup \overline{U}, \wedge) \mapsto m(x)\}_{x \in \mathsf{supp}(m)}$, and Φ_G (defined as above in equation 1) characterises the failure to satisfy $\langle G; \{q_\top\}\rangle$. Notice also that Φ_G contains EU-pair $\langle F; \{(\overline{\mathsf{supp}(F)} \cup \{\overline{g}\}, \wedge)\}\rangle$ where F may contain states of the form $(P, \wedge)$ with $P \subseteq D$, and then $\overline{F}$ may contain states of the form $\overline{(P, \wedge)}$, which we rewrite as $(\overline{P}, \vee)$. This way, δ^c is defined for any state of Q^c and only involves states of Q^c.
  Finally, observe that, as claimed in Remark 7, this construction only introduces new EU-pairs of the form $\langle E; \{u\}\rangle$.
- priorities are defined as follows: $\omega^c(q) = \omega(q)$, $\omega^c(\overline{q}) = \omega(q) + 1$, and $\omega^c(P, op) = \max\{\omega(q) + 1 \mid q \in Q\}$ for all $(P, op) \in Q^c$.

Size of $\mathcal{A}^c$. We now evaluate the size of the complement automaton:

- *(over)approximating the number of states:* all states of $\mathcal{A}^c$ are in $D \cup (2^D \times \{\wedge\}) \cup (2^{(2^D \times \{\vee\})} \times \{\wedge\})$, but not all states of this set are reachable. The reachable states are either in D, or they appear in $\overline{\langle E;U\rangle}$ for some EU-pair $\langle E;U\rangle$ in δ.
  Take an EU-pair $\langle E;U\rangle$ in δ. Besides $(\overline{U},\wedge)$ and $q_\top$, $\overline{\langle E;U\rangle}$ contains, for each submultiset m of E, a formula of the form Φ_G, where $G = (E \setminus m) \uplus m^{\overline{U}}$. Formula Φ_G involves states of G, which are either in E (hence already in Q) or of the form $(\{x\} \cup \overline{U},\wedge)$ for $x \in E$, and states of the form $(\overline{\mathsf{supp}(F)} \cup \{\overline{g}\},\wedge)$ where F is a submultiset of G and $g \in G \setminus F$. In the end, using $|Q| \cdot |\Sigma| \cdot |\delta|_{\mathsf{Bool}}$ as an upper bound on the number of EU-pairs, we have:

$$|Q^c| \leq 2\,|Q| + 1 + |Q| \cdot |\delta|_{\mathsf{Bool}} \cdot |\Sigma| \cdot (1 + |\delta|_{\mathsf{E}} + 2^{|\delta|_{\mathsf{E}}} \cdot |\delta|_{\mathsf{E}})$$

  $Q \cup \overline{Q}$ $q_\top$ number of EU-pairs; state $(\overline{U},\wedge)$; states $(\{x\} \cup \overline{U},\wedge)$; states $(\overline{\mathsf{supp}(F)} \cup \{\overline{g}\},\wedge)$

  Hence $|Q^c| \leq 1 + |Q| \cdot (2 + |\Sigma| \cdot |\delta|_{\mathsf{Bool}} \cdot |\delta|_{\mathsf{E}} \cdot (1 + 2^{|\delta|_{\mathsf{E}}}))$.
- *bounding the size of transition function:* It is easy to see that $|\delta^c_F|_{\mathsf{E}} \leq \max(|\delta|_{\mathsf{E}}, |\delta^c_{\overline{Q}}|_{\mathsf{E}})$ and $|\delta^c_F|_{\mathsf{U}} \leq \max(|\delta|_{\mathsf{U}}, |\delta^c_{\overline{Q}}|_{\mathsf{U}})$. The boolean size is more complex to overapproximate. Intuitively, in the definition of $\overline{\langle E;U\rangle}$, we can bound the number of the subformulas $\Phi_{(E\setminus m)\uplus m^{\overline{U}}}$ by $2^{|\delta|_{\mathsf{E}}}$ and each of them gives rise to a disjunction with at most $|\delta|_{\mathsf{E}} \cdot 2^{|\delta|_{\mathsf{E}}}$ EU-pairs. From this we can see that the boolean size of δ^c from states in $\overline{Q}$ is at most $|\delta|_{\mathsf{Bool}} \cdot (1 + |\delta|_{\mathsf{E}} \cdot (1 + |\delta|_{\mathsf{E}} \cdot 2^{2|\delta|_{\mathsf{E}}}))$. Those formulas also appear in the transitions from states $(\overline{U},\wedge)$ and $(\overline{\mathsf{supp}(F)} \cup \{\overline{g}\},\wedge)$ and states in multisets F (in the definition of Φ_G). Finally we end up with $|\delta^c|_{\mathsf{Bool}} \leq 4 \cdot (1 + |Q|) \cdot |\delta|_{\mathsf{Bool}} \cdot |\delta|_{\mathsf{E}}{}^3 \cdot 2^{2|\delta|_{\mathsf{E}}}$.

  To summarise our results:

▶ **Theorem 11.** *Given an AEUPTA $\mathcal{A} = (Q, q_{init}, \delta, \omega)$, we can build an AEUPTA $\mathcal{A}^c$ recognising the complement of $\mathcal{L}(\mathcal{A})$, with size at most* $(1 + |Q| \cdot (2 + |\Sigma| \cdot |\delta|_{Bool} \cdot (1 + |\delta|_E + 2^{|\delta|_E} \cdot |\delta|_E)), 4 \cdot (1 + |Q|) \cdot |\delta|_{Bool} \cdot |\delta|_E{}^3 \cdot 4^{|\delta|_E}, |\delta|_E + 1, \max(|\delta|_U, 1), |\omega| + 1)$.

3.4 Alternation removal (a.k.a. simulation)

Building a non-alternating automaton equivalent to a given alternating automaton is an important construction, e.g. in order to perform projection, or for algorithmic purposes. In this section, we present an *alternation-removal* (a.k.a. *simulation*) algorithm, based on ideas developed in [30, 33] for MSO-automata. For the rest of this section, we fix an AEUPTA $\mathcal{A} = (Q, q_{init}, \delta, \omega)$. The procedure can be decomposed into four steps:

1. pair each state of the automaton with its immediate ancestor in order to keep enough information in the infinite branches to decide whether they are accepting or not. This provides us with an AEUPTA $\mathcal{P}$ using EU-constraints over $Q \times Q$;
2. build an (alternating) powerset automaton involving a new satisfaction relation $\Vdash$ and a new acceptance condition (which is *not* a parity condition). It gives an AEUTA $\mathcal{Q}$ using EU-constraints over $2^{Q\times Q}$ (and based on $\Vdash$);
3. remove the conjunctions in the transition function of $\mathcal{Q}$ to get a non-alternating automaton. It gives an EUTA $\mathcal{R}$ using EU-constraints over $2^{Q\times Q}$ and relation $\Vdash$;
4. adapt the acceptance condition to obtain an EUPTA $\mathcal{N}$, using EU-constraints over $Q_\omega \times 2^{Q\times Q}$ (where Q_ω is a new set of states introduced to encode the acceptance condition), and relation $\models$.

We briefly describe each step below, and refer to [17, Section 4.4] for more details.

Keeping track of ancestor states. For any state $q' \in Q$, we define the mapping $\phi_{q'} : Q \to Q^2$ as $\phi_{q'}(q) = (q', q)$, and extend it to (multi-)sets of states, EU-pairs and EU-constraints in the natural way. We then define the AEUPTA $\mathcal{P} = (Q^2, (q_{init}, q_{init}), \gamma, \omega')$ with $\gamma((q, q'), \sigma) = \phi_{q'}(\delta(q', \sigma))$, and $\omega'(q, q') = \omega(q')$. Intuitively, state (q, q') in $\mathcal{P}$ corresponds to state q' in $\mathcal{A}$, with the extra information that this state originates from state q. Notice that both γ and ω' only depend only on the second state of the pair (q, q'). We have:

▶ **Proposition 12.** *The languages of $\mathcal{A}$ and $\mathcal{P}$ are equal. The size of $\mathcal{P}$ is $(|Q|^2, |\delta|_{Bool}, |\delta|_E, |\delta|_U, |\omega|)$.*

▶ **Example 13.** Let $\Sigma = \{a, b\}$. Consider an AEUPTA $\mathcal{A}$ with an initial state q_{init} with $\omega(q_{init}) = 1$, and a state q_1 with $\omega(q_1) = 0$, and $\delta(q_{init}, b) = \langle q_1 \mapsto 1; \varnothing\rangle$ and

$$\delta(q_{init}, a) = \langle q_{init} \mapsto 1; \varnothing\rangle \wedge \langle q_1 \mapsto 1; \varnothing\rangle \quad \delta(q_1, a) = \delta(q_1, b) = \langle q_{init} \mapsto 1; \varnothing\rangle.$$

Notice that this automaton only accepts trees with a single branch (i.e., words). It is easily seen that the word $a^3 \cdot b^\omega$ is accepted, while a^ω is not. If we perform a simple powerset construction, the sequence of sets of states along the (unique) computation for both words is $\{q_{init}\} \cdot \{q_{init}, q_1\}^\omega$. This does not keep enough information to decide if a run is accepting. Now, if each state is paired with its ancestor (arbitrarily pairing the initial state with itself), then the sequence of sets of pairs of states visited along $a \cdot b^\omega$ is $\{(q_{init}, q_{init})\} \cdot \{(q_{init}, q_{init}), (q_{init}, q_1)\} \cdot \{(q_1, q_{init}), (q_{init}, q_1)\}^\omega$, while along a^ω it is $\{(q_{init}, q_{init})\} \cdot \{(q_{init}, q_{init}), (q_{init}, q_1)\} \cdot \{(q_{init}, q_{init}), (q_{init}, q_1), (q_1, q_1)\}^\omega$. In the latter sequence, we can detect the presence of an infinite branch looping in q_{init}; we will introduce in the next section a new acceptance condition to capture this fact. ⌟

Building the powerset automaton. In this step, we build an (still alternating) automaton whose states are sets of states of $\mathcal{P}$ (*i.e.* elements of $K = 2^{Q^2}$) with the following changes:

- The satisfaction relation for EU-pairs is modified: instead of considering markings by K, we will consider markings by Q^2 (associating sets of Q-pairs with every successor of a node) from which we will extract submarkings viewed as unitary markings by K in order to satisfy some EU-pair. This is a natural change but it induces a new satisfaction relation denoted $\|\!\models$. The notion of execution tree is then modified accordingly, in particular the execution trees are then labeled by sets of pairs in Q^2.
- The acceptance condition Ω_ω is not a parity condition any more but is defined explicitly.

Formally we define the powerset AEUTA $\mathcal{Q} = (K, \{(q_{init}, q_{init})\}, \beta, \Omega_\omega)$ by letting:

- $K = 2^{Q^2}$ contains all the sets of states of $\mathcal{P}$, hence all the sets of pairs of states of $\mathcal{A}$;
- $\beta\big(\{(q_i, q_i') \mid 1 \le i \le k\}, \sigma\big) = \bigwedge_{1 \le i \le k} \gamma^s((q_i, q_i'), \sigma)$, where $\gamma^s((q_i, q_i'), \sigma)$ is obtained from $\gamma((q_i, q_i'), \sigma)$ by replacing each pair of states (q, q') by the singleton $\{(q, q')\}$. For the time being, this powerset automaton still is alternating.
- the condition Ω_ω works as follows: given a an infinite sequence $\alpha = (\alpha_i)_{i \in \mathbb{N}}$ of subsets of pairs in Q^2 s.t. $\alpha_i = \{(q_{i,j}, q'_{i,j}) \mid 0 \le j \le z_i\}$ for each $i \in \mathbb{N}$, a sequence $(r_i)_{0 \le i < k}$ of states in Q is said to *appear* in α if for each $i \in \mathbb{N}$, there exists an index $0 \le j \le z_i$ such that $(r_{i-1}, r_i) = (q_{i,j}, q'_{i,j})$. The sequence α is *accepting* if all the sequences $(r_i)_{0 \le i < k}$ that appear in α satisfy the parity condition ω of $\mathcal{A}$ (such a sequence α corresponds to the labelling of an infinite branch in the execution trees associated with relation $\|\!\models$).

▶ **Proposition 14.** *A Σ-labelled $\mathcal{D}$-tree $\mathcal{T}$ is accepted by $\mathcal{P}$ if, and only if, it is accepted by the $\mathcal{A}$-powerset AEUTA $\mathcal{Q}$. The size of $\mathcal{Q}$ is $(2^{|Q|^2}, |Q|^2 \cdot |\delta|_{Bool}, |\delta|_E, |\delta|_U, -)$.*

Removing conjunctions. We now remove conjunctions from the transition function β of $\mathcal{Q}$. As a first step, we turn each formula $\beta(P,\sigma)$ in disjunctive normal form. We can bound the number of different EU-pairs appearing in any given $\beta(P,\sigma)$ by $|Q|\cdot|\delta|_{\mathsf{Bool}}$: indeed, while it is built as a conjunction of up to Q^2 transition formulas, any two pairs (q',q) and (q'',q) give rise to the same EU-pairs. It follows that $\beta(P,\sigma)$ can be written as the disjunction of at most $2^{|Q|\cdot|\delta|_{\mathsf{Bool}}}$ conjunctions of at most $|Q|\cdot|\delta|_{\mathsf{Bool}}$ EU-pairs.

We now turn those conjunctions into disjunctions. We proceed inductively, by replacing any conjunction $\langle E_1;U_1\rangle\wedge\langle E_2;U_2\rangle$ of two EU-pairs over $2^{Q\times Q}$ with an "equivalent" disjunction of EU-pairs over $2^{Q\times Q}$.

Write m_1, n_1, m_2 and n_2 for the sizes of E_1, U_1, E_2 and U_2, respectively. The disjunction we build ranges over the possible ways the "existential" and "universal" parts of the EU-pairs overlap (see Fig. 4). For each combination, we write an EU-pair whose existential part contains the "existential" overlaps and the two "mixed" overlaps, and whose universal part handles the "universal" overlap.

The disjunction of EU-pairs can then be written as follows:

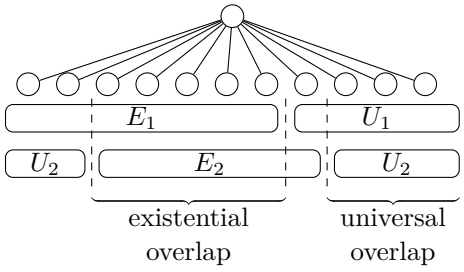


Figure 4 Representation of the overlaps in an execution tree when satisfying a conjunction of two EU-constraints $\langle E_1;U_1\rangle$ and $\langle E_2;U_2\rangle$.

$$C(\langle E_1;U_1\rangle,\langle E_2;U_2\rangle) = \bigvee_{\substack{J_1\sqsubseteq E_1, J_2\sqsubseteq E_2\\ |J_1|=|J_2|}} \bigvee_{\tau \text{ permutation of } [1;|J_1|]} \bigvee_{\substack{g_1:\, E_1\setminus J_1\to U_2\\ g_2:\, E_2\setminus J_2\to U_1}} \left(E' = \biguplus \begin{array}{l} \{\!\{ j^1_k \cup j^2_{\tau(k)} \mid 1\le k\le |J_1| \}\!\} \\ \{\!\{ e^1_k \cup g_1(e^1_k) \mid 1\le k\le n_1-|J_1| \}\!\} \\ \{\!\{ g_2(e^2_k) \cup e^2_k \mid 1\le k\le n_2-|J_2| \}\!\} \end{array} ;U' = U_1\otimes U_2 \right)$$

where we use the notations

$$J_1 = \{\!\{ j^1_k \mid 1\le k\le o \}\!\} \qquad J_2 = \{\!\{ j^2_k \mid 1\le k\le o \}\!\}$$
$$E_1\setminus J_1 = \{\!\{ e^1_k \mid 1\le k\le n_1-o \}\!\} \qquad E_2\setminus J_2 = \{\!\{ e^2_k \mid 1\le k\le n_2-o \}\!\}$$
$$U_1\otimes U_2 = \{u_1\cup u_2 \mid u_1\in U_1, u_2\in U_2\}.$$

Note that the sizes of existential and universal parts of any EU-pair in $C(\langle E_1;U_1\rangle, \langle E_2;U_2\rangle)$ are bounded by n_1+n_2 and $m_1\cdot m_2$, respectively. Note that when U_1 is empty, the only possible overlaps are between E_1 and $\langle E_2;U_2\rangle$ (and we have $J_2=E_2$ and $U_1\otimes U_2$ is also empty). The correctness of the construction is stated as follows (for a proof, see [17]):

▶ **Lemma 15.** *Let $\mathcal{S}$ and $\mathcal{S}'$ be two finite sets, and $\langle E_1;U_1\rangle\wedge\langle E_2;U_2\rangle$ be a conjunction of two EU-pairs on $2^{\mathcal{S}}$. For any unitary marking ν of $\mathcal{S}'$ by $2^{\mathcal{S}}$, it holds $\nu\models\!\!\models \langle E_1;U_1\rangle\wedge\langle E_2;U_2\rangle$ if, and only if, $\nu\models\!\!\models C(\langle E_1;U_1\rangle,\langle E_2;U_2\rangle)$.*

Let $\mathcal{R} = (2^{Q\times Q}, \{(q_{init}, q_{init})\}, \zeta, \Omega_\omega)$ be an $\mathcal{A}$-powerset AEUTA obtained from $\mathcal{Q}$ by replacing all conjunctions $\langle E_1; U_1\rangle \wedge \langle E_2; U_2\rangle$ (in no specific order). We can prove its equivalence with $\mathcal{Q}$ and since $\mathcal{R}$ is non-alternating, it only has to visit each node of the input tree in one of the states given by the transition function, so that both notions of execution trees (with $\models$ and $\|\!\models$) coincide.

We now evaluate the size of $\mathcal{R} = (2^{Q\times Q}, \{(q_{init}, q_{init})\}, \zeta, \Omega_\omega)$. In order to evaluate the size of the transition function ζ of $\mathcal{R}$, we first focus on the size of $C(\langle E_1; U_1\rangle, \langle E_2; U_2\rangle)$. For this, we define the size of an EU-pair $\langle E; U\rangle$ as the pair $(|E|, |U|)$. In the following, a set of EU-pairs is said to have a size at most (n, m) if all its EU-pairs have existential parts of size at most n and universal parts of size at most m.

Consider a conjunction $\langle E_1; U_1\rangle \wedge \langle E_2; U_2\rangle$ of two EU-pairs of size (n_1, m_1) and (n_2, m_2), respectively, and assume w.l.o.g. that $n_1 \leq n_2$. Then the formula $C(\langle E_1; U_1\rangle, \langle E_2; U_2\rangle)$ is a disjunction of N EU-pairs of size at most $(n_1 + n_2, m_1 \cdot m_2)$, with:

$$N \quad \leq \quad \sum_{l=0}^{n_1} \binom{n_1}{l} \cdot \binom{n_2}{l} \cdot l! \cdot m_2^{n_1-l} \cdot m_1^{n2-l} \quad \leq \quad (n_2+1)^{n_1} \cdot m_2^{n_1} \cdot m_1^{n2}.$$

Then we can prove that any conjunction of k EU-pairs of sizes at most (n, m) can be turned into disjunctions of at most $((k-1)! \cdot (n+1)^{k-1} \cdot m^{k^2})^n$ EU-pairs, each of size at most $(k \cdot n, m^k)$ (by induction over k).

We now evaluate the size of the transition function ζ: as explained at the beginning of the present section, ζ is obtained from the transition function β of $\mathcal{Q}$ by first putting each formula $\beta(P, \sigma)$ into disjunctive normal form, as the disjunction of at most $2^{|Q|\cdot|\delta|_{\mathsf{Bool}}}$ conjunctions of at most $|Q| \cdot |\delta|_{\mathsf{Bool}}$ EU-pairs, with EU-pairs of size at most $(|\delta|_{\mathsf{E}}, |\delta|_{\mathsf{U}})$.

Applying our formula above, we get a disjunctive expression for $\beta(P, \sigma)$ involving at most

$$2^{|Q|\cdot|\delta|_{\mathsf{Bool}}} \cdot \left((|Q| \cdot |\delta|_{\mathsf{Bool}} - 1)! \cdot (|\delta|_{\mathsf{E}} + 1)^{|Q|\cdot|\delta|_{\mathsf{Bool}}-1} \cdot (|\delta|_{\mathsf{U}})^{(|Q|\cdot|\delta|_{\mathsf{Bool}})^2}\right)^{|\delta|_{\mathsf{E}}}$$

EU-pairs of size at most $(|Q| \cdot |\delta|_{\mathsf{Bool}} \cdot |\delta|_{\mathsf{E}}, (|\delta|_{\mathsf{U}})^{|Q|\cdot|\delta|_{\mathsf{Bool}}})$. In the end:

▶ **Proposition 16.** *The languages of the original AEUPTA $\mathcal{A}$ and of the resulting EUTA $\mathcal{R}$ are the same. The size of $\mathcal{R}$ is at most $(2^{|Q|^2}, 2^{\mathfrak{K}} \cdot ((\mathfrak{K}-1)! \cdot (|\delta|_E + 1)^{\mathfrak{K}-1} \cdot (|\delta|_U)^{\mathfrak{K}^2})^{|\delta|_E}, \mathfrak{K} \cdot |\delta|_E, (|\delta|_U)^{\mathfrak{K}}, -)$ where $\mathfrak{K} = |Q| \cdot |\delta|_{Bool}$ (we omit the size of the acceptance condition of $\mathcal{R}$ as it is not a parity condition).*

Adapting the acceptance condition. It remains to turn the acceptance condition of $\mathcal{R}$ into a parity condition. The transformation is the same as in [33]: we first build a non-deterministic parity word automaton $\mathcal{W}$ accepting all words on the alphabet $2^{Q\times Q}$ that contain an infinite sequence of states $(r_i)_{i\in\mathbb{N}}$ of $\mathcal{A}$ (in the sense of the definition of $\mathcal{Q}$) *not satisfying* the parity condition of $\mathcal{A}$; we then turn it into a deterministic parity word automaton, take its complement, and run it in parallel with $\mathcal{R}$. The detailed construction is given in [17, Section 4.4.4]. Summarising our results:

▶ **Theorem 17.** *Given an AEUPTA $\mathcal{A} = (Q, q_{init}, \delta, \omega)$, we can build an EUPTA $\mathcal{N}$ recognising the same language. The size of $\mathcal{N}$ can be bounded by $(2^{1+|Q|^2+2(|Q|\cdot|\omega|+1)\cdot\log(|Q|\cdot|\omega|+1)}, 2^{\mathfrak{K}} \cdot ((\mathfrak{K}-1)! \cdot (|\delta|_E + 1)^{\mathfrak{K}-1} \cdot (|\delta|_U)^{\mathfrak{K}^2})^{|\delta|_E}, \mathfrak{K} \cdot |\delta|_E, |\delta|_U{}^{\mathfrak{K}}, 2(|Q| \cdot |\omega| + 1))$, where $\mathfrak{K} = |Q| \cdot |\delta|_{Bool}$.*

4 Algorithms for AEUTAs

Given some AEUTA $\mathcal{A}$, we are interested in two decision procedures: deciding whether a regular tree belongs to $\mathcal{L}(\mathcal{A})$ and deciding whether $\mathcal{L}(\mathcal{A}) = \varnothing$. Both consist in building a parity game and deciding whether Player 0 has a winning strategy and this can be done [4] in time $O(n^d)$ where n is the number of states of the game, and d is the number of priorities [20].

Let $\mathcal{K} = (V, E, \ell)$ be a Kripke structure. Deciding whether $\mathcal{T}_\mathcal{K} \in \mathcal{L}(\mathcal{A})$ is equivalent to deciding whether Player 0 has a winning strategy in the parity game $\mathcal{G}_{\mathcal{A},\mathcal{K}}$ defined after Prop. 8, where the number of states is in $O(|V| \cdot (|Q| \cdot |\delta|_{\mathsf{Bool}} + |Q|^{|V|}))$ and the number of priorities is the same as $\mathcal{A}$. Then we have:

▶ **Theorem 18.** *Given a finite Kripke structure $\mathcal{K} = (V, E, \ell)$ and an AEUPTA $\mathcal{A} = (Q, q_0, \delta, \omega)$, whether $\mathcal{T}_K$ belongs to $\mathcal{L}(\mathcal{A})$ can be decided in time $O((|V| \cdot (|Q| \cdot |\delta|_{Bool} + |Q|^{|V|}))^{|\omega|})$.*

Emptiness checking is performed as for classical tree automata: we consider a non-alternating automaton $\mathcal{A}$, and transform it into a parity game $\mathcal{G}_\mathcal{A}$ such that Player 0 has winning strategy in $\mathcal{G}_\mathcal{A}$ if, and only if, $\mathcal{L}(\mathcal{A}) \neq \varnothing$. It follows:

▶ **Theorem 19.** *Let $\mathcal{A} = (Q, q_0, \delta, \omega)$ be an EUPTA. Checking emptiness of $\mathcal{L}(\mathcal{A})$ can be performed in time $O((|Q| \cdot (1 + |\delta|_{Bool} \cdot |\Sigma|))^{|\omega|})$.*

Combining the previous result with the simulation theorem, we get:

▶ **Corollary 20.** *Let $\mathcal{A} = (Q, q_0, \delta, \omega)$ be an AEUPTA. Checking emptiness of $\mathcal{L}(\mathcal{A})$ can be performed in time $2^{O(|Q|^3 \cdot |\omega| \cdot (\log|Q| + |\delta|_{Bool}{}^2 \cdot |\delta|_E \cdot \log|\delta|_E) + |Q| \cdot |\omega| \cdot \log|\Sigma|)}$.*

5 Application to QCTL and MSO

QCTL extends the temporal logic CTL with quantifications over atomic propositions: subformulas of the form $\exists p.\ \psi$, where p is an atomic proposition, are added to the syntax of CTL (in addition to the usual **EX**, **AX**, **EU** and **AU** modalities, and abbreviations like **EF**, **EG**, **AF** and **AG**). QCTL formulas are interpreted over nodes of 2^{AP}-labelled infinite trees. Informally a node n satisfies $\exists p.\ \psi$ if, and only if, there exists a labelling of the subtree rooted at n for which ψ holds true. This is the so-called *tree semantics* [5]. Given a set $P = \{p_1, \ldots, p_m\} \subseteq \mathsf{AP}$, we write $\exists P.\ \phi$ for $\exists p_1 \ldots \exists p_n.\ \phi$. Propositional quantification can also be added on top of CTL*, yielding the logic QCTL*. We also introduce the sublogics Q^{k}CTL with $k \geq 0$, such that Q^{0}CTL is CTL and Q^{k+1}CTL contains formulas that can be written as $\psi[\exists P_1.\ \phi_1, \ldots, \exists P_n.\ \phi_n]$ where ψ is a CTL formula with placeholders, $P_i \subseteq \mathsf{AP}$, and ϕ_i belongs to Q^{k}CTL for any $1 \leq i \leq n$. The model-checking and satisfiability problems for Q^{k}CTL are known to be k-EXPTIME-complete (resp. $(k+1)$-EXPTIME-complete) [18]. Regarding expressiveness, QCTL (and QCTL*) is as expressive as MSO over infinite trees, hence it is strictly more expressive than CTL (and CTL*).

As we will now explain, our AEUPTA is a type of automata perfectly suited for QCTL (and QCTL*): both formalisms have exactly the same expressiveness, and the automata can be used to derive optimal decision procedures for satisfiability and model-checking.

[4] Better complexity results have been obtained recently for parity games [5], but they would make complexity results even harder to read, without significantly improving them.

[5] It is worth noticing that there exist several semantics for QCTL in the literature [14, 12, 18], with different algorithmic and expressiveness properties.

From QCTL or QCTL* to AEUPTA. We write k-exp(n) to denote the family of sets of functions of one variable n defined inductively as follows: 0-exp(n) is the set of functions bounded by a polynomial in n, and $(k+1)$-exp(n) contains all functions f such that $f \in O(2^g)$ with $g \in k$-exp(n). We have:

▶ **Theorem 21.** *Given a Q^{k}CTL (resp. Q^{k}CTL*) formula ϕ over AP with $k > 0$, we can construct an AEUPTA $\mathcal{A}_\phi$ over 2^{AP} accepting exactly the trees satisfying ϕ. The automaton $\mathcal{A}_\phi$ has size $(k\text{-exp}(|\phi|), k\text{-exp}(|\phi|), (k-1)\text{-exp}(|\phi|), 1, (k-1)\text{-exp}(|\phi|))$ (resp. $((k+1)\text{-exp}(|\phi|), (k+1)\text{-exp}(|\phi|), k\text{-exp}(|\phi|), 1, k\text{-exp}(|\phi|))$).*

We just sketch the main ingredients of the construction for QCTL. First note that any CTL formula ϕ can be turned into an $\{\Box, \Diamond\}$-automata [31]; this construction can easily be adapted to our EU-constraints: $\Diamond q$ can be replaced with $\langle q \mapsto 1; \{q_\top\}\rangle$ and $\Box q$ can be replaced with $\langle \varnothing; \{q\}\rangle$. We then get an AEUPTA $\mathcal{A}_\phi$ whose size is in $(O(|\phi|), O(|\phi|), 1, 1, 2)$.

Now consider a Q^{1}CTL formula ϕ; it can be seen as $\psi[\exists P_1.\ \psi_1, \ldots, \exists P_m.\ \psi_m]$ where ψ is a CTL formula and subformulas ψ_i belong to CTL. We assume w.l.o.g. that negations occur only in front of atomic propositions or $\exists$-quantifiers. From the remark above, we can build an AEUPTA $\mathcal{A}_i$ for each ψ_i with $1 \leq i \leq m$. We handle quantification over P_i in the logic using the projection operation on the automata; this first requires to transform $\mathcal{A}_i$ into an non-alternating automaton $\mathcal{N}_i$ using alternation removal. The size of the resulting automaton (after projection) is in $(1\text{-exp}(|\psi_i|), 1\text{-exp}(|\psi_i|), O(|\psi_i|^2), 1, O(|\psi_i|))$.

Negations in front of $\exists P_i.\ \psi_i$ in the logic are handled with the complementation operation on AEUPTA. It must be noticed that while complementation induces an exponential blow-up in general, this blow-up does not add up with the exponential blow-up for alternation removal. The size of the resulting automaton still is in $(1\text{-exp}(|\psi_i|), 1\text{-exp}(|\psi_i|), O(|\psi_i|^2), 1, O(|\psi_i|))$, Now we can combine all these automata $\mathcal{A}_{\exists P_i.\psi_i}$ (or $\mathcal{A}_{\neg\exists P_i.\psi_i}$) with the automaton for the CTL formula ψ, and get an alternating automaton $\mathcal{A}_\phi$ whose size is the sum of $|\mathcal{A}_\psi|$ and the sizes of automata $|\mathcal{A}_{\exists P_i.\psi_i}|$ (or $|\mathcal{A}_{\neg\exists P_i.\psi_i}|$). In the end, the size of $\mathcal{A}_\phi$ is in $(1\text{-exp}(|\phi|), 1\text{-exp}(|\phi|), O(|\phi|^2), 1, O(|\phi|))$.

The same construction can be applied to any formula ϕ in Q^{k+1}CTL. By induction, we get automata of size $(k\text{-exp}(|\phi|), k\text{-exp}(|\phi|), (k-1)\text{-exp}(|\phi|), 1, (k-1)\text{-exp}(|\phi|))$ for subformulas ψ_i, and end up with an automaton $\mathcal{A}_\phi$ for ϕ whose size is in $((k+1)\text{-exp}(|\phi|), (k+1)\text{-exp}(|\phi|), k\text{-exp}(|\phi|), 1, k\text{-exp}(|\phi|))$.

And back. From an EUPTA $\mathcal{A}$, we can build a QCTL formula $\Phi_\mathcal{A}$ whose size is polynomial in the size of $\mathcal{A}$ and Σ, and such that the models of $\Phi_\mathcal{A}$ are precisely the trees accepted by $\mathcal{A}$. The formula $\Phi_\mathcal{A}$ contains one alternation of quantifiers: it starts wit an existential one, which we use to associate a state of $\mathcal{A}$ with each node of the input tree (as $\mathcal{A}$ is non-alternating, the input tree can be used as accepting tree) and universal quantification is used to ensure the parity condition (see Section 6.3 in [17]). This provides us with an effective translation of any formula $\phi \in$ Q^{k}CTL in Q^{2}CTL, with a $(k+1)$-exp$(|\phi|)$ blow-up in the size of the formula.

Optimal decision procedures. Given a QCTL (res. QCTL*) formula ϕ, one can use the automaton $\mathcal{A}_\phi$ described above in order to decide satisfiability or model-checking. In both cases, the algorithms presented in Section 4 based on the parity games provide optimal algorithms for QCTL and QCTL*. Thus we get the following results (the hardness part is proven in [18]):

▶ **Proposition 22.** *Satisfiability for Q^{k}CTL (resp. Q^{k}CTL*) can be solved in $(k+1)$-EXPTIME (resp. $(k+2)$-EXPTIME) by using the AEUPTA construction and the emptiness checking. Model-checking for Q^{k}CTL (resp. Q^{k}CTL*) can be done in k-EXPTIME (resp. $(k+1)$-EXPTIME) by using the AEUPTA construction and the membership checking.*

Application to MSO. We briefly review *Monadic Second-Order Logic* (*MSO*) over finite or infinite 2^{AP}-labelled trees [25]. MSO is built with first-order variables for vertices (denoted with lowercase letters $x, y, ...$), and monadic second-order variables for sets of vertices (denoted with uppercase letters $X, Y, ...$). Atomic formulas are of the form $x = y$, $\mathsf{Edge}(x, y)$ (to represent the immediate successor relation), $x < y$ (the transitive closure of Edge) $x \in X$, and $\mathsf{P}_a(x)$ for $a \in \mathsf{AP}$. General MSO formulas are constructed from atomic formulas using the boolean connectives and the first- and second-order quantifiers $\exists^1$ and $\exists^2$. We write $\phi(x_1, ..., x_n, X_1, ..., X_k)$ to indicate that $x_1, ..., x_n$ and $X_1, ..., X_k$ may appear free in ϕ. A closed formula contains no free variables. We use the standard semantics for MSO and we write $\mathcal{T}, s_1, ..., s_n, S_1, ..., S_k \models \phi(x_1, ..., x_n, X_1, ..., X_k)$ to indicate that ϕ holds on $\mathcal{T}$ when variables x_1 to x_n in ϕ are replaced with s_1 to s_n, and variables X_1 to X_k are replaced with S_1 to S_k.

In [18], it is proved that MSO and QCTL are equally expressive over trees. This could be used to define translations between MSO and EU-automata, but we can get more efficient constructions as described below. Actually, for any (non-closed) MSO formula $\phi(x_1, ..., x_n, X_1, ..., X_k)$, we build an automaton $\mathcal{A}_\phi$ such that, for any nodes s_1 to s_n and any sets S_1 to S_k, it holds $\mathcal{T}, s_1, ..., s_n, S_1, ..., S_k \models \phi(x_1, ..., x_n, X_1, ..., X_k)$ if, and only if, the $2^{\mathsf{AP} \cup \{x_1,...,x_n,X_1,...,X_k\}}$-labelled tree $\mathcal{T}'$, obtained from $\mathcal{T}$ by labelling any node t with x_i if $t = s_i$, and with X_j if $t \in S_j$, belongs to $\mathcal{L}(\mathcal{A}_\phi)$. Given a closed formula $\Phi \in \mathsf{MSO}$, a 2^{AP}-labelled tree $\mathcal{T}$ belongs to $\mathcal{L}(\mathcal{A}_\Phi)$ if, and only if, $\mathcal{T} \models \Phi$.

Consider a non-closed formula $\phi(x_1, ..., x_n, X_1, ..., X_k)$ where the negation can only be followed by an existential quantifier or a atomic formula. The automaton $\mathcal{A}_\phi$ is built inductively on the structure of ϕ. Boolean connectives $\wedge$ and $\vee$ are handled with the corresponding operations of AEUTA. For quantifications $\exists^1 x.\ \psi$ or $\exists^2 X.\ \psi$, we use projection (after having removed alternation), and check that propositions involved in first-order quantification labels exactly one node of the tree.

Consider a formula in negated normal form (where negations occur only before atomic formula), the complexity of the automata construction depends on the number of (first-order or second-order) quantifier alternations in the formula:

▶ **Theorem 23.** *Given a closed MSO formula ϕ over AP with at most k quantifier alternations (with $k > 0$), we can construct a AEUPTA $\mathcal{A}_\phi$ over 2^{AP} accepting exactly the trees satisfying ϕ. The automaton $\mathcal{A}_\phi$ has size $((k+1)\text{-}exp(|\phi|), (k+1)\text{-}exp(|\phi|), k\text{-}exp(|\phi|), 1, k\text{-}exp(|\phi|))$.*

This allows us to translate any MSO formula with k (first- and second-order) quantifier alternations into a formula with at most four quantifier alternations (and at most one second-order-quantifier alternation) and size in $(k+2)$-exp$(|\phi|)$. Note also that Theorem 23 also provides us with decision procedures in $(k+1)$-EXPTIME for model-checking (and $(k+2)$-EXPTIME for satisfiability).

6 Conclusion

We have introduced a new class of tree automata (AEUPTA) for trees of arbitrary branching degrees. We showed that these automata have exactly the same expressive power as the temporal logics QCTL and QCTL*, and as MSO.

In order to prove those results, we have developed algorithms for manipulating our AEUPTA, and have carefully studied their complexities. This has allowed us to obtain decision procedures for satisfiability and model checking for QCTL* with optimal complexities. It also allowed us to obtain an effective translation from QCTL to Q^{2}CTL, and similarly, from MSO to its fragment with only one second-order-quantifier alternation.

References

1 Orna Bernholtz and Orna Grumberg. Branching time temporal logic and $\mathcal{A}$m**o**r$\textsc{p}\mathcal{H}$0u**s** tree automata. In Eike Best, editor, *Proceedings of the 4th International Conference on Concurrency Theory (CONCUR'93)*, volume 715 of *Lecture Notes in Computer Science*, pages 262–277. Springer-Verlag, 1993.

2 Orna Bernholtz, Moshe Y. Vardi, and Pierre Wolper. An automata-theoretic approach to branching-time model checking (extended abstract). In David L. Dill, editor, *Proceedings of the 6th International Conference on Computer Aided Verification (CAV'94)*, volume 818 of *Lecture Notes in Computer Science*, pages 142–155. Springer-Verlag, 1994. doi:10.1007/3-540-58179-0_50.

3 Mikołaj Bojańczyk. Tree-walking automata. In Carlos Martín-Vide, Friedrich Otto, and Henning Fernau, editors, *Revised Papers of the 2nd International Conference on Language and Automata Theory and Applications (LATA'08)*, volume 5196 of *Lecture Notes in Computer Science*, pages 1–2. Springer-Verlag, 2008. doi:10.1007/978-3-540-88282-4_1.

4 Julius R. Büchi. On a decision method in restricted second order arithmetic. In Ernest Nagel, Patrick Suppes, and Alfred Tarski, editors, *Proceedings of the 1960 International Congress on Logic, Methodology and Philosophy of Science (LMPS'60)*, pages 1–11. Stanford University Press, 1962.

5 Cristian S. Calude, Sanjay Jain, Bakhadyr Khoussainov, Wei Li, and Frank Stephan. Deciding parity games in quasipolynomial time. In Hamed Hatami and Pierre McKenzie, editors, *Proceedings of the 49th Annual ACM Symposium on the Theory of Computing (STOC'17)*, pages 252–263. ACM Press, 2017. doi:10.1145/3055399.305540.

6 Hubert Comon, Max Dauchet, Rémi Gilleron, Florent Jacquemard, Denis Lugiez, Christof Löding, Sophie Tison, and Marc Tommasi. Tree automata techniques and applications. Technical Report hal-03367725, H.A.L., 2008.

7 Arnaud Da Costa, François Laroussinie, and Nicolas Markey. Quantified CTL: Expressiveness and model checking. In Maciej Koutny and Irek Ulidowski, editors, *Proceedings of the 23rd International Conference on Concurrency Theory (CONCUR'12)*, volume 7454 of *Lecture Notes in Computer Science*, pages 177–192. Springer-Verlag, 2012. doi:10.1007/978-3-642-32940-1_14.

8 Calvin C. Elgot. Decision problems of finite automata design and related arithmetics. *Transactions of the American Mathematical Society*, 98(1):21–51, 1961. doi:10.2307/1993511.

9 E. Allen Emerson and Charanjit S. Jutla. Tree automata, mu-calculus and determinacy. In *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science (FOCS'91)*, pages 368–377. IEEE Comp. Soc. Press, 1991. doi:10.1109/SFCS.1991.185392.

10 E. Allen Emerson and A. Prasad Sistla. Deciding full branching time logic. *Information and Control*, 61(3):175–201, 1984. doi:10.1016/S0019-9958(84)80047-9.

11 Tim French. Decidability of quantified propositional branching time logics. In Markus Stumptner, Dan Corbett, and Mike Brooks, editors, *Proceedings of the 14th Australian Joint Conference on Artificial Intelligence (AJCAI'01)*, volume 2256 of *Lecture Notes in Computer Science*, pages 165–176. Springer-Verlag, 2001. doi:10.1007/3-540-45656-2_15.

12 Tim French. *Bisimulation Quantifiers for Modal Logics*. PhD thesis, School of Computer Science & Software Engineering, University of Western Australia, 2006.

13 David Janin and Igor Walukiewicz. Automata for the modal μ-calculus and related results. In Jirí Wiedermann and Petr Hájek, editors, *Proceedings of the 20th International Symposium on Mathematical Foundations of Computer Science (MFCS'95)*, volume 969 of *Lecture Notes in Computer Science*, pages 552–562. Springer-Verlag, 1995. doi:10.1007/3-540-60246-1_160.

14 Orna Kupferman. Augmenting branching temporal logics with existential quantification over atomic propositions. In Pierre Wolper, editor, *Proceedings of the 7th International Conference on Computer Aided Verification (CAV'95)*, volume 939 of *Lecture Notes in Computer Science*, pages 325–338. Springer-Verlag, 1995. doi:10.1007/3-540-60045-0_60.

15 Orna Kupferman, Parthasarathy Madhusudan, P. S. Thiagarajan, and Moshe Y. Vardi. Open systems in reactive environments: Control and synthesis. In Catuscia Palamidessi, editor, *Proceedings of the 11th International Conference on Concurrency Theory (CONCUR'00)*, volume 1877 of *Lecture Notes in Computer Science*, pages 92–107. Springer-Verlag, 2000. doi:10.1007/3-540-44618-4_9.

16 Orna Kupferman and Moshe Y. Vardi. $\Pi_2 \cap \Sigma_2 \equiv$ AFMC. In Jos C. M. Baeten, Jan Karel Lenstra, Joachim Parrow, and Gerhard J. Woeginger, editors, *Proceedings of the 30th International Colloquium on Automata, Languages and Programming (ICALP'03)*, volume 2719 of *Lecture Notes in Computer Science*, pages 697–713. Springer-Verlag, 2003.

17 François Laroussinie and Nicolas Markey. Arbitrary-arity tree automata and QCTL, 2024. doi:10.48550/arXiv.2410.18799.

18 François Laroussinie and Nicolas Markey. Quantified CTL: expressiveness and complexity. *Logical Methods in Computer Science*, 10(4), 2014. doi:10.2168/LMCS-10(4:17)2014.

19 François Laroussinie, Nicolas Markey, and Philippe Schnoebelen. Temporal logic with forgettable past. In *Proceedings of the 17th Annual Symposium on Logic in Computer Science (LICS'02)*, pages 383–392. IEEE Comp. Soc. Press, 2002. doi:10.1109/LICS.2002.1029846.

20 Christof Löding. Automata on infinite trees. In Jean-Éric Pin, editor, *Handbook of automata theory*, volume 1, chapter 8, pages 265–302. EMS Press, 2021. doi:10.4171/AUTOMATA-1/8.

21 Robert McNaughton. Testing and generating infinite sequences by a finite automaton. *Information and Control*, 9(6):521–530, 1966. doi:10.1016/S0019-9958(66)80013-X.

22 David E. Muller and Paul E. Schupp. Alternating automata on infinite objects, determinacy and Rabin's theorem. In Maurice Nivat and Dominique Perrin, editors, *Automata on Infinite Words – École de Printemps d'Informatique Théorique (EPIT'84)*, volume 192 of *Lecture Notes in Computer Science*, pages 99–107. Springer-Verlag, 1985. doi:10.1007/3-540-15641-0_27.

23 Michael O. Rabin. Decidability of second-order theories and automata on infinite trees. *Transactions of the American Mathematical Society*, 141:1–35, 1969.

24 A. Prasad Sistla, Moshe Y. Vardi, and Pierre Wolper. The complementation problem for Büchi automata, with applications to temporal logic. In Wilfried Brauer, editor, *Proceedings of the 12th International Colloquium on Automata, Languages and Programming (ICALP'85)*, volume 194 of *Lecture Notes in Computer Science*, pages 465–474. Springer-Verlag, 1985.

25 Wolfgang Thomas. Languages, automata and logics. In Grzegorz Rozenberg and Arto Salomaa, editors, *Handbook of Formal Languages*, volume 3, pages 389–455. Springer-Verlag, 1997.

26 Boris A. Trakhtenbrot. Finite automata and the logic of one-place predicates. *Siberskii Matematicheskii Zhurnal*, 3(1):103–131, 1962.

27 Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the 1st Annual Symposium on Logic in Computer Science (LICS'86)*, pages 332–344. IEEE Comp. Soc. Press, 1986.

28 Moshe Y. Vardi and Pierre Wolper. Automata theoretic techniques for modal logics of programs. *Journal of Computer and System Sciences*, 32(2):183–221, 1986. doi:10.1016/0022-0000(86)90026-7.

29 Igor Walukiewicz. Monadic second order logic on tree-like structures. In Claude Puech and Rüdiger Reischuk, editors, *Proceedings of the 13th Symposium on Theoretical Aspects of Computer Science (STACS'96)*, volume 1046 of *Lecture Notes in Computer Science*, pages 401–413. Springer-Verlag, 1996. doi:10.1007/3-540-60922-9_33.

30 Igor Walukiewicz. Monadic second order logic on tree-like structures. *Theoretical Computer Science*, 275(1-2):311–346, 2002. doi:10.1016/S0304-3975(01)00185-2.

31 Thomas Wilke. CTL$^+$ is exponentially more succinct than CTL. In C. Pandu Rangan, Venkatesh Raman, and R. Ramanujam, editors, *Proceedings of the 19th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'99)*, volume 1738 of *Lecture Notes in Computer Science*, pages 110–121. Springer-Verlag, 1999. doi:10.1007/3-540-46691-6_9.

32 Thomas Wilke. Alternating tree automata, parity games, and modal μ-calculus. *Bulletin of the Belgian Mathematical Society – Simon Stevin*, 8(2):359–391, 2001.

33 Fabio Zanasi. Expressiveness of monadic second-order logics on infinite trees of arbitrary branching degrees. Master's thesis, Amsterdam University, the Netherlands, 2012.

Galois Energy Games

To Solve All Kinds of Quantitative Reachability Problems

Caroline Lemke
Department of Computing Science, Carl von Ossietzky Universität Oldenburg, Germany

Benjamin Bisping
Institute of Software Engineering and Theoretical Computer Science, Technische Universität Berlin, Germany

Abstract

We provide a generic decision procedure for energy games with energy-bounded attacker and reachability objective, moving beyond vector-valued energies and vector-addition updates. All we demand is that energies form well-founded bounded join-semilattices, and that energy updates have an upward-closed domain and can be "undone" through a Galois-connected function. We instantiate these *Galois energy games* to common energy games, declining energy games, multi-weighted reachability games, coverability on vector addition systems with states, and shortest path problems, supported by an Isabelle-formalization and two implementations. For the instantiations, our simple algorithm is polynomial w.r.t. game graph size and exponential w.r.t. dimension.

2012 ACM Subject Classification Theory of computation

Keywords and phrases Energy games, Galois connection, Reachability, Game theory, Decidability

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2025.29

Related Version *Previous Version*: `https://arxiv.org/abs/2505.14691` [21]
AFP Entry: `https://www.isa-afp.org/entries/Galois_Energy_Games.html` [20]

Supplementary Material *Collection*: `https://github.com/crmrtz/galois-energy-games` [19]

Acknowledgements We want to thank Gabriel Vogel for the WebGPU prototype implementation and Uli Fahrenberg, Lukas Panneke and the anonymous reviewers for their valuable feedback.

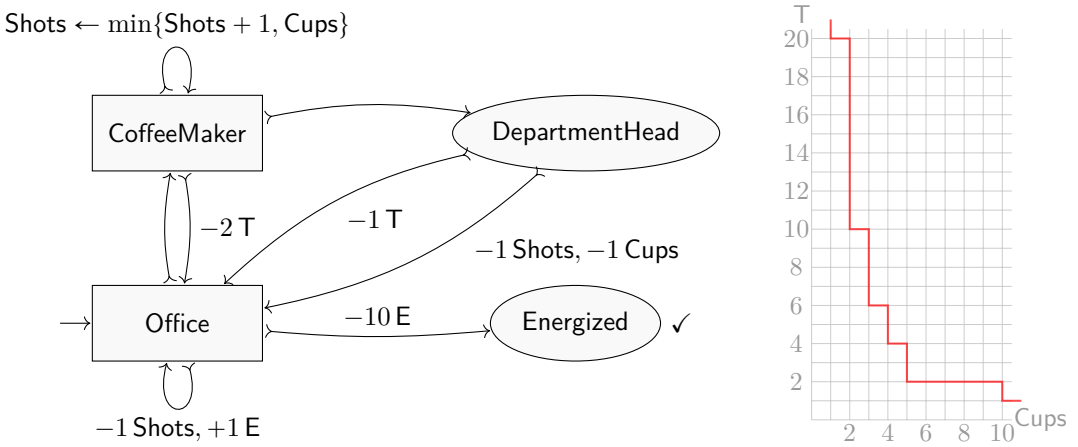
1 Introduction

Most problems of how to reach a goal at minimal cost can be expressed as *energy games* [10, 9]. In energy games, players try to achieve objectives, but fail if they run out of resources that they can gain or lose on the way. But strangely, existing literature does not quite explain how to solve the following kind of multi-weighted energy game:

▶ **Example 1.** The energy game of Figure 1a models the situation of a researcher, who hosts a coauthor at his office and wants to energize her with ten shots of espresso. For this, he leaves his Office for the CoffeeMaker and brews some Shots (bounded by the number of Cups he carries). Upon returning to the Office, the shots can be transformed into energization E. The return usually takes two units of time T. But there is a quicker way, passing by the office of the DepartmentHead. Unfortunately, the department head will either cost our host some time by chatting or help herself to one of the filled coffee cups. (Which of the two happens is not under his control.) Depending on the number of cups, he has to take several rounds until energization E reaches level 10 and the game can be won by moving to Energized.

Assuming that we start with no energization and shots (E = Shots = 0), but with a budget of cups and time, how much of the two does the host need to reach his objective? There are several answers, summarized by the Pareto front of cups and time $\{(1, 20), (2, 10), (3, 6), (4, 4), (5, 2), (10, 1)\}$, which is drawn in Figure 1b.

© Caroline Lemke and Benjamin Bisping;
licensed under Creative Commons License CC-BY 4.0



(a) Game graph. **(b)** Minimal cost of time/cups.

Figure 1 Espresso energy game of Example 1.

The strategy of the host depends on the resources: If there is only one cup, then he should never pass by the department head, where it might be lost. So, he will cycle through $\mathsf{Office} \rightarrowtail \mathsf{CoffeeMaker} \overset{\mathsf{Shots} \leftarrow \min\{\mathsf{Shots}+1,\mathsf{Cups}\}}{\rightarrowtail} \mathsf{CoffeeMaker} \overset{-2\,\mathsf{T}}{\rightarrowtail} \mathsf{Office} \overset{-1\,\mathsf{Shots},+1\,\mathsf{E}}{\rightarrowtail} \mathsf{Office} \ldots$ until the espressos have been served after at least $10 \cdot 2\,\mathsf{T}$. If the host brings 10 or more cups, he can fill them all and safely take the quicker route. There might be a delay at the head's office or it might take a second round, but $1\,\mathsf{T}$ will suffice. For the scenarios in-between, things become more involved ... would it not be nice to have an algorithm to find the minimal budgets there?

Energy games are often studied in light of their closeness to mean-payoff games and (games on) vector addition systems with states [2, 9, 10]. They can model how different resources may interact with each other, for example, in a trade-off between one resource gained (energization) and another lost (espresso shots). Energy updates are usually understood as vector addition. But vector addition is too restricted to capture Example 1, where the Cups component bounds the Shots component!

A recent article by Bisping on process equivalence games [3] introduces *declining energy games*, which allow resources to constrain one-another. But these games prohibit increasing updates! Another recent article by Brihaye and Goeminne [8] formally studies decision procedures and complexity bounds for *multi-weighted reachability games*, which roughly are the intersection of declining and standard energy games, that is, even more restricted.

With this work, we show that we can have all this cake and eat it: Positive as well as negative updates (even multiplicative ones), components bounding each other, and a formalized solution. All we need is that energy updates can be undone through *Galois connections*, a weak form of inversion on monotonic functions. Other than that, we demand little of the precise objects used as energies, which makes our results quite versatile.

Contributions.

- Section 2 introduces *Galois energy games* as a generalization of existing energy game models with an *inductive characterization of winning budgets*.
- Section 3 provides a *simple fixed-point algorithm to compute winning budgets* of Galois energy games and proves its correctness – formalized in Isabelle/HOL.

- We examine how other energy game models and quantitative reachability problems can be *understood as instances of our result* in Section 4 and discuss two prototype implementations.
- We give precise *parametric upper bounds for the complexity* of the algorithm and its instantiations in Section 5: polynomial in game graph size, exponential in dimensionality.
- In Section 6, we discuss how other scenarios and winning conditions could be addressed.

2 Energy Games

Energy games are zero-sum two-player games with perfect information. They are played on directed graphs labeled by energy update functions. Energies represent resources such as time or espresso shots, often modeled as vectors. In our quest for genericity, we model energies as arbitrary partially ordered sets, where each non-empty subset has at least one minimal element and suprema of finite subsets exist, i.e. well-founded bounded join-semilattices. The energy updates describe how each move across an edge affects the current energy, altering the resources as players navigate the game.

2.1 Generic Energy Games

We introduce energy games as generically as possible. For this, we first fix a set of energies $\mathcal{E}$ and assume $(\mathcal{E}, \leqslant)$ to be a well-founded bounded join-semilattice. Further, we assume $(\mathcal{E}, \leqslant)$ to be computable, more precisely we assume $\leqslant$ to be a computable relation and mapping two energies to their supremum to be a computable function. Then, we can define energy games played on directed graphs labeled by updates.

▶ **Definition 2** (Energy game). *An* energy game $\mathcal{G} = (G, G_a, \rightarrowtail)$ *over* $(\mathcal{E}, \leqslant)$ *consists of*

- *a set of positions* G*, partitioned into attacker positions* $G_a \subseteq G$ *and defender positions* $G_d := G \backslash G_a$*, and*
- *a partial function* $\rightarrowtail : (G \times G) \to (\mathcal{E} \to \mathcal{E})$ *mapping edges to partial energy functions.*

For $g, g' \in G$ *we write* $g \overset{u}{\rightarrowtail} g'$ *instead of* $\rightarrowtail (g, g') = u$ *and call* u *an update in* $\mathcal{G}$*. The game with initial position* $g_0 \in G$ *and initial energy* $e_0 \in \mathcal{E}$ *is denoted as* $\mathcal{G}[g_0, e_0]$*.*

▶ **Example 3.** Formally, Definition 2 does not permit the two edges between DepartmentHead and Office in Example 1. However, this can be fixed by adding a node Chat such that DepartmentHead $\overset{-1\,\mathsf{T}}{\rightarrowtail}$ Chat $\rightarrowtail$ Office. In the example, the researcher getting coffee is the attacker, choosing at rectangular positions, while the environment is the defender at elliptical positions. The resources (cups, time, shots, energization) can be represented as four-dimensional natural vectors, making Example 1 an energy game over $\mathbb{N}^4$ with the component-wise order.

In an energy game, the players move one token from one position to the next according to the edges, resulting in a play. At attacker positions, the attacker chooses (according to some attacker strategy) and vice versa.

▶ **Definition 4** (Play, strategy). *Let* $\mathcal{G} = (G, G_a, \rightarrowtail)$ *be an energy game over* $(\mathcal{E}, \leqslant)$*. A* play *in* $\mathcal{G}$ *is a possibly infinite walk* $\rho = g_0 g_1 \ldots \in G^* \cup G^\omega$ *in the underlying directed graph.*

Let $G_p \in \{G_a, G_d\}$*. A* strategy $s : G^* G_p \to G$ *is a partial function mapping finite plays to a next position such that* $s(g_0 \ldots g_n)$ *is some successor of* g_n *if* $g_n \in G_p$ *is not a deadlock. If* $G_p = G_a$ *(resp.* $G_p = G_d$*) we call* s *an attacker (resp. defender) strategy. A play* $\rho = g_0 g_1 \ldots$ *is consistent with* s *if* $g_i \in G_p$ *implies* $g_{i+1} = s(g_0 \ldots g_i)$ *for each* $i \in \mathbb{N}$ *with* $i + 1 < |\rho|$*.*

During a play, an initial energy is repeatedly updated according to the edges in the play. This is defined as the energy level, which keeps track of the resources. The attacker wins a play if and only if it ends in a deadlocked defender position before running out of energy, i.e. without the energy level becoming undefined.

▶ **Definition 5** (Energy level, winning a play). *Let $\mathcal{G} = (G, G_a, \rightarrowtail)$ be an energy game over $(\mathcal{E}, \leqslant)$. The* energy level *of a play $\rho = g_0 g_1 \dots$ w.r.t. initial energy $e_0 \in \mathcal{E}$ is inductively defined as $EL(\rho, e_0, 0) := e_0$ and $EL(\rho, e_0, i+1) := u(EL(\rho, e_0, i))$ where $g_i \overset{u}{\rightarrowtail} g_{i+1}$ and $i + 1 < |\rho|$.*

If the energy level becomes undefined or the play is infinite, the defender wins the play. If the final energy level $EL(\rho, e_0, |\rho| - 1)$ of a finite play is defined and its last position g_l is a deadlock, then the defender wins the play if $g_l \in G_a$ and the attacker wins if $g_l \in G_d$.

In other settings, infinite plays (where the energy level is always defined) are won by the energy-bound player, i.e. our attacker [10, 15]. We will revisit this distinction in Section 6 as safety winning conditions. Now we define what it means to win a game w.r.t. a starting position and an initial energy.

▶ **Definition 6** (Winning strategy, winning a game). *Let $\mathcal{G} = (G, G_a, \rightarrowtail)$ be an energy game over $(\mathcal{E}, \leqslant)$. For $g_0 \in G$ and $e_0 \in \mathcal{E}$, an attacker winning strategy for $\mathcal{G}[g_0, e_0]$ is an attacker strategy ensuring the attacker to eventually win all consistent plays.*

The attacker wins the game $\mathcal{G}[g_0, e_0]$ if and only if such an attacker winning strategy exists. The attacker winning budget *of a position $g \in G$ is the set of energies sufficient for the attacker to enforce a win, i.e. $\mathsf{Win}_a(g) = \{e \in \mathcal{E} \mid \exists s.\ s \text{ is an attacker winning strategy for } \mathcal{G}[g, e]\}$.*

Defender winning strategies and budgets can be defined symmetrically to Definition 6. However, it is sufficient to focus on the attacker's objective, since energy games are determined w.r.t. a starting position and initial energy.

▶ **Lemma 7.** *Let $\mathcal{G}$ be an energy game over $(\mathcal{E}, \leqslant)$. For all positions $g \in G$ and energies $e \in \mathcal{E}$ either the defender or the attacker wins $\mathcal{G}[g, e]$. Further, there exists an energy-positional winning strategy, i.e. a winning strategy that depends only on the current position and energy level.*

Lemma 7 follows from the construction of a parity game, which is positionally determined. For more details we refer to the appendix.

▶ Remark 8. The winning conditions in Definition 5 and 6 encode quantitative reachability problems: In reachability games [9], the attacker aims to reach a fixed target set in a (weighted) game graph. By treating defender deadlock positions as the target set, we formulate (quantitative) reachability winning conditions.

When starting with a multi-weighted reachability game [8], i.e. a game $(G, G_a, \rightarrowtail)$ weighted by vectors of naturals with a target set $F \subseteq G$, this can easily be transformed to an energy game: Remove outgoing edges of positions in $G_d \cap F$ and add a loop to every deadlock in $G_d \backslash F$. Further, add a deadlocked defender position with an ingoing edge labeled by the zero-vector from each position in $G_a \cap F$. By this construction, a defender position is a deadlock if and only if it corresponds to a position in the target set F in the original game. In the reachability game, one is interested in the minimal ensured cost for a position g, i.e. a minimal upper bound that the attacker can enforce while winning for the sum of edge weights of plays starting in g. By switching from addition to subtraction, this corresponds to asking for the minimal attacker winning budget of g in the constructed energy game.

2.2 Inductive Characterization of Attacker Winning Budgets

We want to decide energy games by calculating the attacker winning budgets. For this, we study attacker winning budgets and observe that they are upward-closed in monotonic energy games, i.e. where all updates are monotonic and have an upward-closed domain.

▶ **Definition 9** (Monotonic energy game). *Let $\mathcal{G}$ be an energy game over $(\mathcal{E}, \leqslant)$. Then, $\mathcal{G}$ is a* monotonic energy game *if, for all updates u in $\mathcal{G}$ and all $e, e' \in \mathcal{E}$ with $e \leqslant e'$, the following implication holds: If $u(e)$ is defined, then so is $u(e')$ and $u(e) \leqslant u(e')$.*

▶ **Lemma 10.** *Let $\mathcal{G}$ be a monotonic energy game over $(\mathcal{E}, \leqslant)$. Then, the attacker winning budgets are upward-closed, i.e. $\mathsf{Win}_a(g) = {\uparrow}\,\mathsf{Win}_a(g) := \{e \in \mathcal{E} \mid \exists e' \in \mathsf{Win}_a(g).\ e' \leqslant e\}$ for all positions $g \in G$.*

Lemma 10 follows directly from the definition of winning budgets and monotonic games since the same strategy serves as a witness when starting with more energy.

In monotonic energy games, each attacker winning budget of a position g can be characterized by a finite Pareto front, i.e. by the minimal attacker winning budget $\mathsf{Win}_a^{\min}(g) := \operatorname{Min} \mathsf{Win}_a(g)$. (For a set M we distinguish between the set of all minimal elements, i.e. $\operatorname{Min} M := \{m \in M \mid \forall m' \in M.\ m' \not\leqslant m\}$, and the minimum. We write $\min M$ to denote the minimum if it exists.) Since the energies are well-founded, antichains in $(\mathcal{E}, \leqslant)$ are finite and thereby so are the sets $\mathsf{Win}_a^{\min}(g)$ for all positions g.

Attacker winning budgets have more structure. In particular, they can be characterized inductively.

▶ **Lemma 11.** *Let $\mathcal{G}$ be an energy game over $(\mathcal{E}, \leqslant)$. The attacker winning budgets can be characterized inductively by the rules:*

$$\frac{g \in G_a \qquad g \overset{u}{\rightarrowtail} g' \qquad u(e) \in \mathsf{Win}_a(g')}{e \in \mathsf{Win}_a(g)} \qquad \frac{g \in G_d \qquad \forall g'.\ g \overset{u}{\rightarrowtail} g' \longrightarrow u(e) \in \mathsf{Win}_a(g')}{e \in \mathsf{Win}_a(g)}$$

Proof. An attacker winning strategy can be constructed from attacker winning strategies at successors w.r.t. the accordingly updated energies. This implies soundness.

To argue completeness we utilize Lemma 7. Assuming $e \in \mathsf{Win}_a(g)$, there exists an energy-positional attacker winning strategy s for $\mathcal{G}[g, e]$. We then fix an order induced by s by setting $(g'', e'') \leqslant_s (g', e')$ if $(g'', e'') = (g', e')$ or there is an edge in the configuration graph from (g', e') to (g'', e'') consistent with s, i.e. if $g' \overset{u}{\rightarrowtail} g''$, $e'' = u(e')$ and $g' \in G_a \longrightarrow s(g', e') = g''$. Then, a derivation of $e \in \mathsf{Win}_a(g)$ can be found by well-founded induction. ◀

2.3 Galois Energy Games

In order to convert the inductive characterization of Lemma 11 into a fixed-point algorithm, it seems that we need to *invert* energy updates in the game. However, it turns out that what is sufficient is a weaker form of invertibility given by *Galois connections*.

▶ **Definition 12** (Galois energy game). *A* Galois energy game *is a monotonic energy game $\mathcal{G}$ over $(\mathcal{E}, \leqslant)$ such that $u^{\mathcal{O}}(e') := \min\{e \in \mathcal{E} \mid e' \leqslant u(e)\}$ exists and is computable for all updates u in $\mathcal{G}$ and $e' \in \mathcal{E}$.*

Galois connections can be characterized by one of the following four equivalent properties. We refer to Erné et al. [14] for more detail.

▶ **Lemma 13.** *Let $\mathcal{P} = (P, \leqslant_P)$ and $\mathcal{Q} = (Q, \leqslant_Q)$ be partially ordered sets with functions $f : P \to Q$ and $g : Q \to P$. Then, the following are equivalent:*

1. *$f(p) \leqslant_Q q \longleftrightarrow p \leqslant_P g(q)$ holds for all $p \in P$ and $q \in Q$.*
2. *f and g are monotonic, $g \circ f$ is increasing, and $f \circ g$ is decreasing.*
3. *f is monotonic and $g(q) = \max_P\{p \in P \mid f(p) \leqslant_Q q\}$ for each $q \in Q$.*
4. *g is monotonic and $f(p) = \min_Q\{q \in Q \mid p \leqslant_P g(q)\}$ for each $p \in P$.*

If any of the (and thereby all) properties hold, then f and g form a Galois connection between P and Q. By Lemma 13 an energy game $\mathcal{G}$ over $(\mathcal{E}, \leqslant)$ is a Galois energy game if and only if each update u in $\mathcal{G}$ has an upward-closed domain and there exists a computable function $u^{\circlearrowleft} : \mathcal{E} \to \mathrm{dom}(u)$ such that $u^{\circlearrowleft}$ and u form a Galois connection between $\mathcal{E}$ and $\mathrm{dom}(u)$. Then, the minima used to define $u^{\circlearrowleft}$ in Definition 12 always exist.

▶ **Example 14.** The energy game from Example 1 is a Galois energy game. In particular, $(\mathsf{Shots} \leftarrow \min\{\mathsf{Shots} + 1, \mathsf{Cups}\})^{\circlearrowleft}(e_\mathsf{C}, e_\mathsf{T}, e_\mathsf{S}, e_\mathsf{E}) = (\max\{e_\mathsf{C}, e_\mathsf{S}\}, e_\mathsf{T}, \max\{e_\mathsf{S} - 1, 0\}, e_\mathsf{E})$.

3 Deciding Galois Energy Games

For a fixed position g, the *known initial credit problem* asks whether energy e is sufficient for the attacker to always win when starting from g, i.e. whether $e \in \mathsf{Win}_a(g)$. The *unknown initial credit problem* asks, whether there exists an energy sufficient for the attacker to win from g, i.e. whether $\mathsf{Win}_a(g) \neq \varnothing$. Subsuming both these problems, we provide an algorithm calculating the *minimal attacker winning budgets of each position.*

The core idea of the algorithm is that of a shortest path algorithm: We start at deadlocks and move backwards through the graph using the inductive characterization of attacker winning budgets (Lemma 11). To calculate energies backwards, we use Galois connections, allowing us to prove:

▶ **Theorem 15** (Decidability of Galois energy games)**.** *Let $\mathcal{G}$ be a Galois energy game over $(\mathcal{E}, \leqslant)$ with a finite set of positions. Then, the (un)known initial credit problem is decidable.*

3.1 The Algorithm

To prove Theorem 15, we first discuss Algorithm 1. This is a simplified and generalized version of an algorithm proposed by Bisping [3].

Algorithm 1 Computing minimal attacker winning budgets of Galois energy game $\mathcal{G}$.

```
1  def compute_winning_budgets(𝒢 = (G, G_a, ↦)):
2      win := [g ↦ {} | g ∈ G]
3      do
4          old_win := win
5          for g ∈ G :
6              if g ∈ G_a :
7                  win[g] := Min{u^↻(e') | g ↣^u g' ∧ e' ∈ old_win[g']}
8              else:
9                  win[g] := Min{sup{u^↻(e_g')|g ↣^u g'} | ∀g'. g ↣^u g' ⟶ e_g' ∈ old_win[g']}
10     while win ≠ old_win
11     return old_win
```

Starting by assigning the empty set to each position in line 2, we then apply lines 3 to 10 adding sufficient energies to winning budgets and taking the minimal elements (which exist by well-foundedness) until we reach a fixed point to return.

For a play to be won by the attacker, it has to end in a defender deadlock position. At such a defender position g_d all energies are in the attacker winning budget and $\mathsf{Win}_a^{\min}(g_d) = \operatorname{Min} \mathcal{E}$. If a defender position has successors, then an energy is in the attacker winning budgets if it is sufficient to win moving to any successor the defender might choose. In Galois energy games, that is $e \in \mathsf{Win}_a(g_d)$ for $g_d \in G_d$ if $u^{\mathcal{O}}(e_{g'}) \leqslant e$ for some $e_{g'} \in \mathsf{Win}_a(g')$ for each g' with $g_d \rightarrowtail^u g'$. This is ensured by line 9. Note that the supremum always exists because the energies form a bounded join-semilattice and the sets in question are finite.

Similarly, sufficient energies for attacker positions can be calculated from energies in the winning budget of successors, as done in line 7. In particular, $g_a \in G_a$ and $e' \in \mathsf{Win}_a(g')$ for some g' with $g_a \rightarrowtail^u g'$ implies $u^{\mathcal{O}}(e') \in \mathsf{Win}_a(g_a)$.

Table 1 Application of Algorithm 1 to Example 1.

Iteration	0	1	2	3	...
Energized	$\varnothing$	$\{(0,0,0,0)\}$	$\{(0,0,0,0)\}$	$\{(0,0,0,0)\}$	...
Office	$\varnothing$	$\varnothing$	$\{(0,0,0,10)\}$	$\{(0,0,0,10),(0,0,1,9)\}$	...
CoffeeMaker	$\varnothing$	$\varnothing$	$\varnothing$	$\{(0,2,0,10)\}$	...

Table 1 illustrates how the information, which positions are winnable, travels through the graph when applying Algorithm 1 to Example 1. The vectors represent cups, time, shots and energization in that order.

3.2 Correctness

Since we are considering a fixed point algorithm, we use Kleene's fixed point theorem [25, 28] to prove correctness of our algorithm and thereby our decidability result, Theorem 15. The following version of Kleene's fixed point theorem relies on the notion of directed sets, i.e. a subset of a partially ordered set where all pairs of elements have an upper bound in the subset. In a directed-complete partial order all directed sets have a (directed) supremum. A Scott-continuous function preserves directed suprema.

▶ **Lemma 16** (Kleene's fixed point theorem). *Let $(Q, \leqslant_Q)$ be a directed-complete partial order with least element $q \in Q$ and a Scott-continuous function $f : Q \to Q$. Then, the set of fixed points of f in Q forms a complete lattice with least fixed point $\sup_Q\{f^i(q) \mid i \in \mathbb{N}\}$, where f^i refers to the i-th iterate of f.*

Proof of Theorem 15. To utilize Kleene's fixed point theorem, we define a partial order on mappings from positions to possible Pareto fronts, i.e. antichains in $\mathcal{E}$. We call the set of such mappings $\mathsf{Pareto}_{\mathcal{G}} \subseteq (G \to 2^{\mathcal{E}})$ and define

$$F \preceq F' \longleftrightarrow \forall g \in G.\ F(g) \subseteq \uparrow F'(g) \quad \text{and} \quad (\sup\nolimits_{\preceq} P)(g) := \operatorname{Min} \bigcup_{F \in P} \uparrow F(g)$$

for $F, F' \in \mathsf{Pareto}_{\mathcal{G}}$ and $P \subseteq \mathsf{Pareto}_{\mathcal{G}}$. These definitions yield that $(\mathsf{Pareto}_{\mathcal{G}}, \preceq)$ is a directed-complete partial order with minimal element $\mathbf{0} := \lambda g.\ \varnothing$.

We then consider the function $\mathsf{Iteration}_{\mathcal{G}} : \mathsf{Pareto}_{\mathcal{G}} \to \mathsf{Pareto}_{\mathcal{G}}$ corresponding to one iteration of the while-loop in Algorithm 1 when applied to $\mathcal{G}$ and show that it is Scott-continuous w.r.t. the $\preceq$-order, i.e. $\mathsf{Iteration}_{\mathcal{G}}(\sup_{\preceq} P) = \sup_{\preceq}\{\mathsf{Iteration}_{\mathcal{G}}(F) \mid F \in P\}$ for all directed sets $P \subseteq \mathsf{Pareto}_{\mathcal{G}}$.

Let $P \subseteq \mathsf{Pareto}_{\mathcal{G}}$ be such a directed set. The definition of $\mathsf{Iteration}_{\mathcal{G}}$ and monotonicity of all $u^{\mathcal{O}}$ together imply monotonicity of $\mathsf{Iteration}_{\mathcal{G}}$ and thereby $\mathsf{Iteration}_{\mathcal{G}}(F) \preceq \mathsf{Iteration}_{\mathcal{G}}(\sup_{\preceq} P)$ for all $F \in P$. Therefore, what remains to show is $\mathsf{Iteration}_{\mathcal{G}}(\sup_{\preceq} P) \preceq \sup_{\preceq}\{\mathsf{Iteration}_{\mathcal{G}}(F) \mid F \in P\}$. Let $g \in G$ and $e \in \mathsf{Iteration}_{\mathcal{G}}(\sup_{\preceq} P)(g)$. We now show $e \in \uparrow (\sup_{\preceq}\{\mathsf{Iteration}_{\mathcal{G}}(F) \mid F \in P\})(g)$. We focus on the more intricate case and assume $g \in G_d$. By definition of $\mathsf{Iteration}_{\mathcal{G}}$, there exist $e_{g'} \in (\sup_{\preceq} P)(g')$ for each $g' \in G$ with $g \rightarrowtail^{u} g'$ such that $e = \sup\{u^{\mathcal{O}}(e_{g'}) \mid g \rightarrowtail^{u} g'\}$. By definition of $\sup_{\preceq}$, there exist $F_{g'} \in P$ with $e_{g'} \in \uparrow F_{g'}(g')$. Since the set of positions is finite, so is $\{F_{g'} \mid g \rightarrowtail^{u} g'\} \subseteq P$ and there exists an upper bound $F' \in P$ such that $e_{g'} \in \uparrow F'(g')$ for each successor g' of g. Thus, $e \in \uparrow \mathsf{Iteration}(F')(g) \subseteq \uparrow (\sup_{\preceq}\{\mathsf{Iteration}(F) \mid F \in P\})(g)$.

With this, we can apply Lemma 16 and conclude that a least fixed point, namely $\sup_{\preceq}\{\mathsf{Iteration}^{i}_{\mathcal{G}}(\mathbf{0}) \mid i \in \mathbb{N}\}$, exists. We now show, that the least fixed point is $\mathsf{Win}^{\min}_{a}$ by first arguing that $\mathsf{Win}^{\min}_{a}$ is a fixed point. For this, we introduce inductively-defined sets $S(g)$ for $g \in G$ mimicking the process of the iteration of the while-loop without taking the minimum:

$$\frac{g \in G_a \qquad g \rightarrowtail^{u} g' \qquad e' \in S(g')}{u^{\mathcal{O}}(e') \in S(g)} \qquad \frac{g \in G_d \qquad \forall g'.\ g \rightarrowtail^{u} g' \longrightarrow e_{g'} \in S(g')}{\left(\sup\{u^{\mathcal{O}}(e_{g'}) \mid g \rightarrowtail^{u} g'\}\right) \in S(g)}$$

Then, $S(g) \subseteq \mathsf{Win}_a(g)$ for all $g \in G$ follows from a simple induction over the structure of S, utilizing the inductive characterization of attacker winning budgets as well as Lemma 10 and 13. Further, $\mathsf{Win}_a(g) \subseteq \uparrow S(g)$ can be shown by a well-founded induction over the strategy-induced order introduced in the proof of Lemma 11 using Lemma 13. With Lemma 10, we can conclude $\mathsf{Win}_a(g) = \uparrow S(g)$ and thereby $\mathsf{Win}^{\min}_{a} = \mathrm{Min} \circ S$. Further, the definition of S directly implies that $\mathrm{Min} \circ S$ (and therefore $\mathsf{Win}^{\min}_{a}$) is a fixed point of $\mathsf{Iteration}_{\mathcal{G}}$. For any fixed point F of $\mathsf{Iteration}_{\mathcal{G}}$, we have $S(g) \subseteq \uparrow F(g)$ for all $g \in G$ by induction over the structure of S. Thus, $\mathrm{Min} \circ S$ (and therefore $\mathsf{Win}^{\min}_{a}$) is the least fixed point of $\mathsf{Iteration}_{\mathcal{G}}$.

Finally, we argue for termination by noting that, for each $g \in G$ and $e \in \mathsf{Win}^{\min}_{a}(g)$, there exists $i^{e}_{g} \in \mathbb{N}$ such that e is added in the i^{e}_{g}-th iteration at the latest, i.e. $e \in \mathsf{Iteration}^{i^{e}_{g}}_{\mathcal{G}}(\mathbf{0})(g)$. This holds true since $\mathsf{Win}^{\min}_{a} = \sup_{\preceq}\{\mathsf{Iteration}^{i}_{\mathcal{G}}(\mathbf{0}) \mid i \in \mathbb{N}\}$. Thus, a fixed point is reached after at most $\max_{g \in G} \max_{e \in \mathsf{Win}^{\min}_{a}(g)} i^{e}_{g}$ iterations of the while-loop. ◀

The proof of Theorem 15 also proves that without any computability assumptions Algorithm 1 is still correct, i.e. the least fixed point of $\mathsf{Iteration}_{\mathcal{G}}$ exists and is $\mathsf{Win}^{\min}_{a}$.

3.3 Isabelle/HOL Formalization

The proof of decidability of Galois energy games (Theorem 15) is formalized in Isabelle/HOL. Isabelle [23] is a generic interactive proof assistant supporting the formalization of mathematical theories. Theorems formalized in Isabelle undergo automated verification, ensuring that edge cases are not overlooked. While Isabelle supports multiple logical frameworks, we use the instantiation Isabelle/HOL based on Higher Order Logic (HOL). The Isabelle theories are available at `https://github.com/crmrtz/galois-energy-games`.

Our Isabelle formalization starts by providing the first formalization of this kind for any type of energy game. This includes the inductive characterization of attacker winning budgets (Lemma 11) assuming energy-positional determinacy (Lemma 7). Fixing an arbitrary Galois energy game over a well-founded bounded join-semilattice, we then formalize the function $\mathsf{Iteration}_{\mathcal{G}}$ introduced in the proof of Theorem 15. We formalize the correctness of Algorithm 1 by showing that the minimal attacker winning budgets are the least fixed point of $\mathsf{Iteration}_{\mathcal{G}}$ (correctness) and show termination. Thus, we have formalized the decidability of such Galois

energy games. Finally, we formalize Galois energy games over vectors of (extended) naturals with the component-wise order and conclude decidability, particularly for energy games with vector addition and taking minima of components only.

Table 2 Overview of the Isabelle/HOL formalization.

	Result	Formalization link	Proof adjustments
Lemma 7	Energy-positional determinancy	-	
Lemma 10	Upwards-closure of Win_a	`upward_closure_wb_nonpos`	
Lemma 11	Inductive characterization of Win_a	`inductive_winning_budget`	assuming Lemma 7
Lemma 13	Galois connection equivalences	`galois_properties`	only 1. ⟶ 2.
Lemma 16	Kleene's fixed point theorem	by Yamada and Dubut [28]	
Theorem 15	Decidability	Correctness: `a_win_min_is_lfp` Termination: `finite_iterations`	assuming Lemma 7

Table 2 provides an overview of all previously discussed lemmas and theorems, along with information on which ones have been formalized and links to their formalization. For the instances discussed in Section 4 the formalization links are given in footnotes.

4 Instances of Galois Energy Games

Let us discuss how Galois energy games and the algorithm can be used to solve quantitative problems by instantiating energy orders and updates over vectors of naturals.

For the illustration, we will take an extremely classical problem and two recent ones. In Subsection 4.1, we show how shortest paths problems on graphs are encoded as single-player energy games and relate the latter to vector addition systems with states (VASS). As more recent problems, Subsection 4.2 will discuss the declining energy games of Bisping [3] and the multi-weighted reachability games of Brihaye and Goeminne [8], in a generalized version.

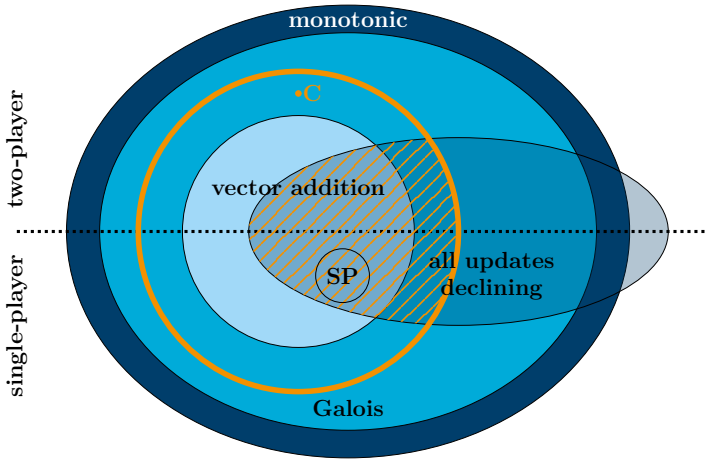


Figure 2 Energy games over vectors of naturals with the component-wise order.

Figure 2 lays out the kinds of energy games to be discussed: The yellow circle includes all energy games with updates that mix positive, negative, and min-updates, in particular, our example energy game of the researcher getting coffee placed as yellow **C**. The crosshatched area includes the "declining energy games" discussed by Bisping [3], where the subclass with vector addition only corresponds to the games discussed by Brihaye and Goeminne [8]. The

bottom half-circle of vector addition contains energy games solving the coverability problem on VASS. **SP** in that set refers to the single-player energy games corresponding to shortest path problems for the one-dimensional case as will be discussed in Subsection 4.1.

The class of monotonic energy games is decidable too. For this, Lemke [19] generalizes Algorithm 1 by calculating sets of the form $\mathrm{Min}\{e \in \mathcal{E} \mid e' \leqslant u(e)\}$ if a unique minimum does not exist, which adds another level of combinatorial explosion. For details on the boundaries of decidability for monotonic games, we refer to Abdulla et al. [1]. We focus our attention on Galois energy games, which generalize previous results in a simple yet concrete manner.

4.1 Singe-Player Instances

Single-player energy games are energy games, where all defender positions are deadlocks and, therefore, only the attacker makes choices.

Shortest Paths and Distances

Shortest path and shortest distance problems [22] are classical quantitative problems. These can be seen as instances of single-player energy games, where the minimal attacker winning budgets correspond to a kind of *shortest distances* from a position to any defender position. Let us illustrate this for the one-dimensional case:

▶ **Proposition 17.** *Assume we have a graph (V, E) with $E \subseteq V \times \mathbb{N} \times V$, and ask for the length of shortest paths from some source node $s \in V$ to target node $t \in V$. Consider the energy game $(V \cup \{\bot\}, V, \rightarrowtail)$ over $(\mathbb{N}, \leqslant)$ with $v \overset{-w}{\rightarrowtail} v'$ if $(v, w, v') \in E$ and $t \overset{0}{\rightarrowtail} \bot$. Then, $e \in \mathsf{Win}_a^{\min}(s)$ on the derived game precisely if e is the shortest distance from s to t in the original graph.*

On this instance, our Algorithm 1 boils down to a single-destination variant of the Bellman–Ford algorithm for shortest paths.

▶ Remark 18. Mixing positive and negative weights, the construction of Proposition 17 encounters differences to classical shortest-path solutions, which assume commutativity of edge costs. For example, in the graphs $s \xrightarrow{2} \circ \xrightarrow{-1} t$ and $s \xrightarrow{-1} \circ \xrightarrow{2} t$, Bellman–Ford will consider both s-t-paths to have length 1. However, the energy game interpretation leads to a different result: Moving $\overset{-2}{\rightarrowtail}\!\!\!\!\twoheadrightarrow\overset{1}{\longrightarrow}$ requires an attacker budget of 2, while $\overset{1}{\rightarrowtail}\!\!\!\!\twoheadrightarrow\overset{-2}{\longrightarrow}$ only needs 1. In some cases the energy interpretation is more accurate in reality: If one has an empty tank, it makes a difference whether one can refuel now or only later.

Also, conventional shortest-path algorithms struggle with negative cycles, which correspond to positive cycles in our games. In the energy interpretation, we easily determine the cost to reach a target even with an infinite charging cycle: The cost of getting to the cycle.

Vector Addition Systems with States

The single-player version of a multi-dimensional energy game over $\mathbb{N}^n$ with the component-wise order $\leqslant_c$ where edges are labeled by functions adding a vector in $\mathbb{Z}^n$ can be used to answer the *coverability problem* [17] on VASS. In VASS, this problem asks whether a target position can be reached from a given initial configuration (a pair of a position and an energy) with at least a fixed energy remaining.

▶ **Proposition 19.** *Let $n \in \mathbb{N}$. Assume we have a VASS (Q, T) with $T \subseteq Q \times \mathbb{Z}^n \times Q$ with a fixed initial configuration $(s, e_s) \in Q \times \mathbb{N}^n$ and a target configuration $(t, e_t) \in Q \times \mathbb{N}^n$. Consider the energy game $(Q \cup \{\bot\}, Q, \rightarrowtail)$ over $(\mathbb{N}^n, \leqslant_c)$ with $v \overset{w}{\rightarrowtail} v'$ if $(v, w, v') \in T$ and $t \overset{-e_t}{\rightarrowtail} \bot$. Then, $e_s \in \mathsf{Win}_a(s)$ on the derived game precisely if coverability holds in the original VASS.*

The *reachability problem* [6] on VASS aks whether a configuration is reachable from another. Since the target energy has to be reached exactly, this is a stricter version of coverability. As in Proposition 19, this can be modeled as an energy game by labeling the edge $t \rightarrowtail \bot$ with a function to check if the current energy matches the target (that is undefined if not). However, since such a function lacks an upward-closed domain, our results are not applicable and we refer to existing literature [6, 12, 13] instead.

4.2 Energy Games over Vectors of Naturals

In recent research, energy games with reachability winning conditions have garnered some attention [3, 8]. We now demonstrate that we generalize their results. We do so with a syntactical approach to define a set of updates permitted in Galois energy games over vectors of naturals. We introduce the set $\mathcal{U}_n$ of possible updates that allow changing entries in two ways: By adding an integer or replacing it with the minimum of components. Such updates can be represented as vectors where each entry contains the information how the respective component is updated. First, we fix a set of energies: The set of vectors of extended naturals $\mathbb{N}_\infty^n$ of dimension $n \in \mathbb{N}$ with the component-wise order $\leqslant_c$.

▶ **Definition 20.** *Let $n \in \mathbb{N}$ and $\mathcal{U}_n := \left(\mathbb{Z} \cup 2^{\{0,\dots,n-1\}} \setminus \{\varnothing\}\right)^n$. Then, we interpret $u = (u_0, \dots, u_{n-1}) \in \mathcal{U}_n$ as a partial function $u : \mathbb{N}_\infty^n \to \mathbb{N}_\infty^n$ by setting*

$$u(e_0, \dots, e_{n-1}) := (e'_0, \dots, e'_{n-1}) \text{ with } e'_i := \begin{cases} e_i + z, & \text{if } u_i = z \in \mathbb{Z} \wedge -z \leqslant e_i \\ \min_{k \in D} e_k, & \text{if } u_i = D \subseteq \{0, \dots, n-1\} \end{cases}$$

if all e'_i are defined.

As a consequence of Theorem 15, we obtain the following.

▶ **Corollary 21.** *Let $n \in \mathbb{N}$ and let $\mathcal{G} = (G, G_a, \rightarrowtail)$ be an energy game over $(\mathbb{N}_\infty^n, \leqslant_c)$ with updates in $\mathcal{U}_n$. Then, $\mathcal{G}$ is a Galois energy game. Further, if the set of positions is finite, the game is decidable.*[1]

Proof. We argue that $\mathcal{G}$ is a Galois energy game: By definition, all updates in $\mathcal{U}_n$ are monotonic with an upward-closed domain. Generalizing the inversion in [3], we show that $u^{\circlearrowleft}(e') = \min\{e \mid e' \leqslant u(e)\}$ exists for all $u \in \mathcal{U}_n$ and $e' \in \mathcal{E}$ by giving a computable calculation.

Let u be an update in $\mathcal{G}$, i.e. $u = (u_0, \dots, u_{n-1}) \in \mathcal{U}_n$. For $e' = (e'_0, \dots, e'_{n-1}) \in \mathbb{N}_\infty^n$ we set $u^{\circlearrowleft}(e')$ such that the i-th component is the maximum of

- $e'_i - z$, if this is not negative and the update adds z in the i-th component,
- e'_j, if the i-th component is used when taking the minimum resulting in the j-th component,
- 0 else.

That is

$$u^{\circlearrowleft}(e')_i := \max(\{e'_i - z \mid u_i = +z \in +\mathbb{Z} \wedge z \leqslant e'_i\} \cup \{e'_j \mid i \in u_j \subseteq \{0, \dots, n-1\}\} \cup \{0\}). \blacktriangleleft$$

▶ Remark 22. All our arguments for $\mathcal{U}_n$ still hold, if we allow multiplication with a nonzero natural. When calculating $u^{\circlearrowleft}(e')_i$ we then have to add $\left\lceil \frac{e'_i}{m} \right\rceil$ to the maximum, if the update multiplies the i-th component with m.

[1] This is formalized using Remark 25 to establish a subclass relationship, i.e. a `sublocale` relationship.

▶ **Example 23.** Note that all occurring updates in Example 1 aside from the loop at CoffeeMaker are elements of $\mathcal{U}_4$. That loop can be replaced by adding a Brew-position such that CoffeeMaker $\rightarrowtail^{+1\ \mathsf{Shots}}$ Brew $\rightarrowtail^{\mathsf{Shots}\leftarrow\min\{\mathsf{Shots},\mathsf{Cups}\}}$ CoffeeMaker.

Actually, the adjustment of the game graph described in Example 23 is not necessary to prove the game to be a Galois energy game given that Galois connections compose.

▶ **Lemma 24.** *Let $u_1, u_2 : \mathcal{E} \to \mathcal{E}$ be partial functions with upward-closed domains such that $u_i^{\circlearrowleft}$ and u_i forms a Galois connection between $\mathcal{E}$ and $\mathrm{dom}(u_i)$ for $i \in \{1, 2\}$. Then, $(u_2 \circ u_1)^{\circlearrowleft} := u_1^{\circlearrowleft} \circ u_2^{\circlearrowleft}$ and $u_2 \circ u_1$ form a Galois connection between $\mathcal{E}$ and the reverse image under u_1 of $\mathrm{dom}(u_2)$, i.e. $u_1^{-1}(\mathrm{dom}(u_2))$.*[2]

▶ Remark 25. Lemma 24 implies that it suffices to study simple updates as generators and check that they are permitted in a Galois energy game, to prove decidability. In particular, we may focus on updates in $\{-1, 0, +1\}^n \subseteq \mathbb{Z}^n$ that add or subtract at most one in any component to understand all updates in $\mathbb{Z}^n \subseteq \mathcal{U}_n$.

Now we can turn to instances where multi-weighted energy games with reachability conditions were considered. Bisping et al. [3, 4, 5] introduce games generalizing the bisimulation game to simultaneously decide all common notions of behavioral equivalence, i.e. the linear-time-branching-time spectrum [26]. In particular, they use energy games over $(\mathbb{N}_\infty^n, \leqslant_c)$ with declining updates in $\mathcal{U}_n$. The decidability of those games is a direct implication of Corollary 21.

The multi-weighted reachability games described by Brihaye and Goeminne [8] can easily be transformed to energy games with declining updates in $\mathbb{Z}^n$ by utilizing the construction from Remark 8. When considering the component-wise order, decidability follows directly from Corollary 21. When considering the lexicographical order $\leqslant_l$, decidability can be concluded using Theorem 15.

4.3 Implementations

We now illustrate the practical feasibility of Algorithm 1, which has been implemented twice in open source software: Once, in the *Linear-Time–Branching-Time Spectroscope*, a web tool to decide spectra of behavioral equivalence [4], and once in *gpuequiv*, a GPU-accelerated version thereof [27]. Both work on energy games of arbitrary-dimensional vectors of naturals.

The *Spectroscope* is implemented in Scala.js, powering the tool on `https://equiv.io/`. Its energy game class supports positive, negative, and min-updates. Example 1 is reproduced as a unit test in `io.equiv.eqfiddle.game.GaloisEnergyGameTests`.

gpuequiv [27] uses Rust and the WebGPU system to solve declining energy games on the GPU. The core is an implementation of our algorithm split up across several shaders in order to exploit GPU parallelism in `gpuequiv::energygame::EnergyGame`. For the shader implementation, it is particularly important that the buffer size for the Pareto fronts can be bounded, as will be discussed in Section 5. (In order to conserve memory, the shader will initially reserve space for fronts of up to 64 entries during updates, and must retry positions with more space whenever the allocated space does not suffice.) So far, gpuequiv only supports declining energy games, for which it includes several tests. According to benchmarking [27, Chapter 5], *gpuequiv* tends to solve the games 10 to 20 times faster than the Scala implementation (even if the latter is compiled and run through JVM instead of JS).

Both implementations increase our confidence that the approach works well in practical scenarios, and hopefully can be of help to others who want to solve similar problems.

[2] This is formalized as `galois_composition`.

5 Complexity

The complexity of deciding Galois energy games varies based on the energy order and the updates. Since deciding multi-weighted reachability games is PSPACE-complete as stated by Brihaye and Goeminne [8], deciding energy games over vectors of naturals with the component-wise order and only vector-subtraction is PSPACE-hard:

▶ **Proposition 26.** *Deciding Galois energy games over n-dimensional vectors of naturals with the component-wise order is PSPACE-hard with respect to n.*

In this section, we give more fine-grained upper complexity bounds for the generic algorithm and for the instantiations according to the previous section.

The running time of Algorithm 1 depends on factors beyond input size, such as the occurring updates which affect the number of iterations needed. To analyze the influence of the order on complexity, we introduce functions that will help establish upper bounds for attacker winning budgets, which are then used in our complexity results.

▶ **Definition 27.** *For $e \in \mathcal{E}$ let $\mathsf{antichain}_{\leqslant e}$ be the set of antichains in $\{e' \in \mathcal{E} \mid e' \leqslant e\}$. Then, $hgt(e)$ is the cardinality of $\{e' \in \mathcal{E} \mid e' \leqslant e\}$ and $wdh(e)$ is the maximal cardinality of elements in $\mathsf{antichain}_{\leqslant e}$, i.e. $wdh(e) := \sup_{\mathbb{N}}\{|A| \mid A \in \mathsf{antichain}_{\leqslant e}\}$.*

Fixing even more parameters, we can state our complexity result.

▶ **Theorem 28** (Complexity). *Let $\mathcal{G} = (G, G_a, \rightarrowtail)$ be a Galois energy game over $(\mathcal{E}, \leqslant)$. Further, let*

- *o be the branching degree of the underlying graph,*
- *$t_{\sup}$ be an upper bound of the time to compute the supremum of two energies,*
- *$t_{\leqslant}$ be an upper bound of the time to compute the comparison of two energies,*
- *$t_{\circlearrowleft}$ be an upper bound of the time to compute $u^{\circlearrowleft}(e)$ for any update u in $\mathcal{G}$ and $e \in \mathcal{E}$, and*
- *$e_{worst} := \sup\{u_1^{\circlearrowleft} \circ \ldots \circ u_i^{\circlearrowleft}(\min \mathcal{E}) \mid i \in \{0, \ldots, |G|-1\} \wedge u_1, \ldots, u_i$ are updates in $\mathcal{G}\}$ be the highest energy obtainable using up to $|G|-1$ reverse updates applied to the least element of $\mathcal{E}$.*

Then, Algorithm 1 on input $\mathcal{G}$ terminates in a time in

$$\mathcal{O}\left(|G|^2 \cdot o \cdot hgt(e_{worst}) \cdot wdh(e_{worst})^2 \cdot (t_{\circlearrowleft} + t_{sup} + wdh(e_{worst}) \cdot t_{\leqslant})\right),$$

and in $\mathcal{O}\left(|G|^2 \cdot o \cdot wdh(e_{worst})^2 \cdot (t_{\circlearrowleft} + t_{sup} + wdh(e_{worst}) \cdot t_{\leqslant})\right)$, if $\mathcal{G}$ is declining.

In both cases, the output is calculated using space in $\mathcal{O}\left(|G| \cdot wdh(e_{worst}) \cdot s_{\mathcal{E}}\right)$ where $s_{\mathcal{E}}$ is an upper bound for the space needed to store any energy.

For a detailed proof, we refer to the appendix. Compared to the approach by Bisping [3], this result significantly improves complexity – see Table 3. The key argument leading to this improvement is the calculation of line 9 of Algorithm 1, i.e. $\mathsf{win}[\mathsf{g}]$ for a defender position g. Inspired by Brihaye and Goeminne [8], we apply the following procedure:

```
1 def compute_new_win(𝒢, old_win, g):
2     new := Min ℰ
3     for g′ with g ⇀^u g′:
4         new := Min{sup{e′, u^↺(e_g′)} | e′ ∈ new ∧ e_g′ ∈ old_win[g′]}
5     return new
```

Table 3 compares the complexity results we obtain by Theorem 28 to results stated by Bisping [3] on declining energy games and Brihaye and Goeminne [8] on multi-weighted reachability games, respectively. Further, it states the complexity of Algorithm 1 when applied to a shortest path problem as discussed in Proposition 17 and for energy games with updates in $\mathbb{Z}^n \subseteq \mathcal{U}_n$, i.e. integer vector addition, subsuming the complexity of solving coverability on VASS as seen in Proposition 19.

To do so, we first give upper bounds for some variables: In all cases, we over-approximate o with $|G|$ when applying Theorem 28. Further, we assume $t_{\leqslant_l}, t_{\leqslant_c}, t_{\sup} \in \mathcal{O}(n)$. We use w as an upper bound of the absolute value of the integers being added to a component at any edge.[3] Then, we have $e_{worst} = w \cdot (|G|, \dots, |G|)$ and $\mathrm{hgt}(e_{worst}) \in \mathcal{O}\left((w \cdot |G|)^n\right)$. Other upper bounds are stated in Table 3, where all entries are to be understood in big $\mathcal{O}$-notation.

Table 3 Application of Theorem 28 to energy games over vectors of naturals.

Game model	$t_{\mathcal{O}}$	$\mathrm{wdh}(e_{worst})$	time complexity	complexity in cited work
Declining energies [3]	n^2	$\lvert G\rvert^{n-1}$	$n^2 \cdot \lvert G\rvert^{3n}$	$\lvert\rightarrowtail\rvert \cdot \lvert G\rvert^n \cdot (o + \lvert G\rvert^{(n-1)o})$
Multi-reachability, component-wise $\leqslant_c$ [8]	n	$w^{n-1} \cdot \lvert G\rvert^{n-1}$	$n \cdot w^{3(n-1)} \cdot \lvert G\rvert^{3n}$	$n^4 \cdot w^{4n} \cdot \lvert G\rvert^{4n+1}$
Multi-reachability, lexicographic $\leqslant_l$ [8]	n	1	$n \cdot \lvert G\rvert^3$	$n \cdot \lvert G\rvert^3$
Shortest paths (with $\mathbb{N}$-edges)	1	1	$\lvert G\rvert^3$	.
Integer vector addition, component-wise $\leqslant_c$	n	$w^{n-1} \cdot \lvert G\rvert^{n-1}$	$n \cdot w^{4n-3} \cdot \lvert G\rvert^{4n}$	.

Due to similarities in both the algorithm and approximation techniques, it is not surprising that our complexity result (stated in Table 3) for multi-weighted reachability games with the lexicographic order coincides with previous work by Brihaye and Goeminne [8]. The improvement, when considering the component-wise order, can be attributed primarily to a finer approximation. However, we do improve Bisping's [3] result (in the case of a branching degree greater than two). This comparison demonstrates that our algorithm achieves a competitive performance while offering generality.

6 How to Handle Other Winning Conditions

Reachability winning conditions are fundamental to the analysis of games, both in their simplicity and their importance. In this section, we examine how other types of winning conditions relate to reachability (Definition 5 and 6) studying related games.

6.1 Within the Scope: Captured Games

Weak Upper Bounds. The defender winning if the energy level becomes undefined is another way of saying that there is a lower bound for the energy (and the attacker is energy-bound). We now consider energy games with an additional *upper bound* [15, 16] of resources the attacker may hold. A *weak upper bound* caps the energies. In Example 1, the number of cups resembles a weak upper bound of the number of shots. Similarly, a weak upper bound can be added to any energy game, where the minimum is monotonic, by adding a new dimension for each dimension that should be bounded and composing every update with the function taking the minimum of the dimension and its upper bound. Then, Algorithm 1 can be used in the case of both, a known and an unknown weak upper bound.

[3] To get a fair comparison we consider the complexity of declining energy games as formally defined by Bisping [3] where an update may subtract at most one from a component, i.e. $w = 1$.

Generalized Reachability. *Generalized reachability* [9] refers to reachability games with multiple target sets where the attacker wins a play if each set has been visited at least once (without running out of energy). Combining the construction in Remark 8 and that for weak upper bounds, an energy game can model a generalized reachability game by encoding the target sets using additional dimensions that track the visited sets. For this construction, we add positions and edges such that the attacker may choose after reaching any target set, whether to continue or to move to the only deadlock defender position at the cost of one in every dimension corresponding to a target set. Note that this construction increases the dimension and thereby increases time complexity of our algorithm for computing minimal winning budgets exponentially in the number of target sets.

6.2 At the Borders: Related but Distinct Games

Strong Upper Bounds. We reconsider energy games with an upper bound of the attackers resources. Exceeding a *strict upper bound* [16] results in the defender winning. Such an upper bound can be integrated similarly to a weak upper bound by adding dimensions for the bounded dimensions and instead of taking the minimum adding the difference between the energy and the updated energy in the dimensions corresponding to the bounds at every edge. (Inspired by Juhl et al. [16] we additionally apply the mapping $b_i \mapsto b_i + (e_i - u(e)_i)$ to each added bound b_i for the bounded dimension i when $e \mapsto u(e)$ is calculated.) However, if the updates are more complex than vector addition, this construction might not yield a Galois energy game – the game might not even be monotonic.

Safety Winning Conditions. Instead of aiming to reach a target set, the goal of *safety* [1] winning conditions is to avoid a set of unsafe positions. This corresponds to changing the winner of infinite plays to the attacker (if the energy level does not become undefined). Proving decidability then is symmetric to the proofs presented in this paper and can be achieved by adjusting the definitions and statements accordingly. In particular, this includes calculating maximal defender winning budgets instead and defining Galois energy games using Lemma 13.3, i.e. as monotonic energy games where $u^{\circlearrowleft}(e') := \max\{e \in \mathcal{E} \uplus \{\bot\} \mid u(e) \leqslant e'\}$ always exists and is computable. (Here $\bot$ represents undefined energies with $\bot < e$ for all $e \in \mathcal{E}$ and $u(\bot) := \bot$.)

Parity Games. An *energy parity game* [2, 24] is an energy game with a ranking function that assigns each position a natural number (called priority) and has a finite image. Then, infinite plays where the energy level never becomes undefined are won by the attacker if and only if the lowest priority appearing infinitely often is even. A stronger objective for the attacker would be to never visit more than l positions with odd priority before visiting a position with a smaller even priority for a fixed $l \in \mathbb{N}$. Abdulla et al. [2] show that deciding games with that stronger objective in some cases is equivalent to deciding energy parity games. Such energy parity games with vector addition only can be formulated as an energy game with safety winning conditions as described by Chatterjee et al. [11] where dimensions corresponding to the odd priorities are added. Whether such a procedure can be applied to other kinds of updates remains an open question.

6.3 Beyond the Horizon: Games Outside the Framework

Both-Bounded Energy Games. In *both-bounded energy games* [18] both players are energy-bound. This is modeled by assigning two updates to each edge, operating on pairs of current energies – one for the attacker, one for the defender. Considering such games over vectors of

naturals, undecidability arises if one player has one-dimensional energies and the other two, so our results cannot be transferred.

Payoffs of Plays. To compare infinite plays and decide a winner, different *payoffs* [7] can be calculated. The *mean-payoff* refers to the limit average gain per edge, i.e. $\limsup_{n\to\infty} \frac{1}{n} EL(\rho, e_0, n)$ for a play ρ w.r.t. an initial energy e_0, while the *total-playoff* is the limit energy level, i.e. $\limsup_{n\to\infty} EL(\rho, e_0, n)$, and the *average-energy* refers to the limit average energy level, i.e. $\limsup_{n\to\infty} \frac{1}{n} \sum_{i=0}^{n} EL(\rho, e_0, i)$. Since Algorithm 1 relies on the fact that it suffices to consider finite plays, our results cannot be transferred directly.

7 Conclusion

In this paper, we presented a simple algorithm for calculating minimal winning budgets, solving the (un)known initial credit problem for energy games over any well-founded bounded join-semilattice. We generalized two recent approaches [3, 8] and improved the running time. Our results demonstrate that Galois connections offer a simple framework for constructing decidable energy games. Compared to Abdulla et al.'s monotonic games [1] or to Mohri's generalized shortest-distances [22], our work is more concrete in its application. For a relevant class of energy games over vectors of naturals, we offer an algorithm implementation for practical use. Additionally, we provide an Isabelle/HOL formalization of the general decidability proof, ensuring confidence in our results. By exploring various winning conditions and relating games to Galois energy games, we highlighted the theory's aplicability and limitations and identified areas for further research, such as using Galois connections to classify energy parity games.

References

1 Parosh Aziz Abdulla, Ahmed Bouajjani, and Julien d'Orso. Deciding monotonic games. In *Computer Science Logic (CSL)*, volume 2803 of *Lecture Notes in Computer Science*, pages 1–14. Springer Berlin Heidelberg, 2003. doi:10.1007/978-3-540-45220-1_1.

2 Parosh Aziz Abdulla, Richard Mayr, Arnaud Sangnier, and Jeremy Sproston. Solving parity games on integer vectors. In *Concurrency Theory (CONCUR)*, volume 8052 of *Lecture Notes in Computer Science*, pages 106–120. Springer Berlin Heidelberg, 2013. doi:10.1007/978-3-642-40184-8_9.

3 Benjamin Bisping. Process equivalence problems as energy games. In *Computer Aided Verification (CAV)*, volume 13964 of *Lecture Notes in Computer Science*, pages 85–106. Springer Nature Switzerland, 2023. doi:10.1007/978-3-031-37706-8_5.

4 Benjamin Bisping. *Generalized Equivalence Checking of Concurrent Programs*. PhD thesis, Technische Universität Berlin, 2025. URL: https://generalized-equivalence-checking.equiv.io/.

5 Benjamin Bisping and David N. Jansen. One energy game for the spectrum between branching bisimilarity and weak trace semantics. In Georgiana Caltais and Cinzia Di Giusto, editors, Proceedings Combined 31st International Workshop on *Expressiveness in Concurrency* and 21st Workshop on *Structural Operational Semantics,* Calgary, Canada, 9th September 2024, volume 412, pages 71–88. Open Publishing Association, 2024. doi:10.4204/EPTCS.412.6.

6 Michael Blondin, Matthias Englert, Alain Finkel, Stefan Göller, Christoph Haase, Ranko Lazic, Pierre McKenzie, and Patrick Totzke. The reachability problem for two-dimensional vector addition systems with states. *Journal of the ACM*, 68(5):32–43, 2021. doi:10.1145/3464794.

7 Patricia Bouyer, Nicolas Markey, Mickael Randour, Kim G Larsen, and Simon Laursen. Average-energy games. *Acta Informatica*, 55:91–127, 2018. doi:10.1007/s00236-016-0274-1.

8 Thomas Brihaye and Aline Goeminne. Multi-weighted reachability games. In *Reachability Problems*, volume 14235 of *Lecture Notes in Computer Science*, pages 85–97. Springer Nature Switzerland, 2023. doi:10.1007/978-3-031-45286-4_7.

9 Thomas Brihaye, Aline Goeminne, James C. A. Main, and Mickael Randour. Reachability games and friends: A journey through the lens of memory and complexity. In Patricia Bouyer and Srikanth Srinivasan, editors, *43rd IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, volume 284 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1:1–1:26, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.FSTTCS.2023.1.

10 Krishnendu Chatterjee, Laurent Doyen, Thomas A. Henzinger, and Jean-François Raskin. Generalized Mean-payoff and Energy Games. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, volume 8 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 505–516, Dagstuhl, Germany, 2010. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.FSTTCS.2010.505.

11 Krishnendu Chatterjee, Mickael Randour, and Jean-François Raskin. Strategy synthesis for multi-dimensional quantitative objectives. In *Concurrency Theory (CONCUR)*, volume 7454 of *Lecture Notes in Computer Science*, pages 115–131. Springer Berlin Heidelberg, 2012. doi:10.1007/978-3-642-32940-1_10.

12 Wojciech Czerwiński, Ismaël Jecker, Sławomir Lasota, and Łukasz Orlikowski. Reachability in 3-vass is elementary. *arXiv preprint arXiv:2502.13916*, 2025. doi:10.48550/arXiv.2502.13916.

13 Wojciech Czerwiński and Łukasz Orlikowski. Reachability in vector addition systems is ackermann-complete. In *2021 IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1229–1240, 2021. doi:10.1109/FOCS52979.2021.00120.

14 Marcel Erné, Jürgen Koslowski, Austin Melton, and George E. Strecker. A primer on galois connections. *Annals of the New York Academy of Sciences*, 704(1):103–125, 1993. doi:10.1111/j.1749-6632.1993.tb52513.x.

15 Uli Fahrenberg, Line Juhl, Kim G. Larsen, and Jiří Srba. Energy games in multiweighted automata. In Antonio Cerone and Pekka Pihlajasaari, editors, *Theoretical Aspects of Computing (ICTAC)*, pages 95–115. Springer Berlin Heidelberg, 2011. doi:10.1007/978-3-642-23283-1_9.

16 Line Juhl, Kim Guldstrand Larsen, and Jean-François Raskin. *Optimal Bounds for Multi-weighted and Parametrised Energy Games*, pages 244–255. Springer Berlin Heidelberg, 2013. doi:10.1007/978-3-642-39698-4_15.

17 Marvin Künnemann, Filip Mazowiecki, Lia Schütze, Henry Sinclair-Banks, and Karol Węgrzycki. Coverability in vass revisited: Improving rackoff's bound to obtain conditional optimality. In *50th International Colloquium on Automata, Languages, and Programming (ICALP 2023)*, volume 261 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 131:1–131:20, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ICALP.2023.131.

18 Orna Kupferman and Naama Shamash Halevy. Energy games with resource-bounded environments. In *Concurrency Theory (CONCUR)*, volume 243 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 19:1–19:23, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CONCUR.2022.19.

19 Caroline Lemke. crmrtz/galois-energy-games – A formal proof of decidability of multi-weighted declining energy games. Master's thesis, Technische Universität Berlin, 2024. Collection (visited on 2025-08-05). URL: https://github.com/crmrtz/galois-energy-games, doi:10.4230/artifacts.24329.

20 Caroline Lemke. Galois energy games. *Archive of Formal Proofs*, 2025. Formal proof development. URL: https://isa-afp.org/entries/Galois_Energy_Games.html.

21 Caroline Lemke and Benjamin Bisping. Galois energy games: To solve all kinds of quantitative reachability problems, 2025. doi:10.48550/arXiv.2505.14691.

22 Mehryar Mohri. Semiring frameworks and algorithms for shortest-distance problems. *Journal of Automata, Languages and Combinatorics*, 7(3):321–350, 2002. doi:10.25596/jalc-2002-321.

23 Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2002. doi:10.1007/3-540-45949-9.

24 Sven Schewe, Alexander Weinert, and Martin Zimmermann. Parity games with weights. *Logical Methods in Computer Science*, 15, 2019. doi:10.23638/LMCS-15(3:20)2019.

25 Joseph E Stoy. *Denotational semantics : the Scott-Strachey approach to programming language theory*. The MIT Press series in computer science. The MIT Press, 3. print edition, 1981.

26 Rob J. van Glabbeek. The linear time-branching time spectrum (extended abstract). In Jos C. M. Baeten and Jan Willem Klop, editors, *Theories of Concurrency: Unification and Extension (CONCUR)*, volume 458 of *Lecture Notes in Computer Science*, pages 278–297. Springer Berlin Heidelberg, 1990. doi:10.1007/BFb0039066.

27 Gabriel Vogel. Accelerating process equivalence energy games using WebGPU. Bachelor's thesis, Technische Universität Berlin, 2024. URL: https://github.com/Gobbel2000/gpuequiv.

28 Akihisa Yamada and Jérémy Dubut. Complete non-orders and fixed points. *Archive of Formal Proofs*, 2019. URL: https://www.isa-afp.org/entries/Complete_Non_Orders.html.

A Appendix

This appendix contains the proofs of selected results from the main body of the paper.

Energy-positional Determinacy

▶ **Lemma 7.** *Let $\mathcal{G}$ be an energy game over $(\mathcal{E}, \leqslant)$. For all positions $g \in G$ and energies $e \in \mathcal{E}$ either the defender or the attacker wins $\mathcal{G}[g, e]$. Further, there exists an energy-positional winning strategy, i.e. a winning strategy that depends only on the current position and energy level.*

Proof. The (energy-positional) determinacy follows from the positional determinacy of parity games. We define a parity game $\mathcal{P}_\mathcal{G} := (G_\mathcal{G}, G_{\mathcal{G}0}, E_\mathcal{G}, r_\mathcal{G})$ where

- $G_\mathcal{G} := \{(g, e) \mid g \in G, e \in \mathcal{E} \uplus \{\bot\}\}$,
- $G_{\mathcal{G}0} := G_\mathcal{G} \backslash G_{\mathcal{G}1}$ with $G_{\mathcal{G}1} := \{(g, e) \in G_\mathcal{G} \mid g \in G_a \vee e = \bot\}$,
- $E_\mathcal{G} := \{((g, e), (g', e')) \in G_\mathcal{G} \times G_\mathcal{G} \mid g \rightarrowtail^u g' \wedge e \neq \bot \wedge u(e) = e'\}$ and
- $r_\mathcal{G} : G_\mathcal{G} \to \mathbb{N}, (g, e) \mapsto 0$.

This construction is similar to that of a configuration graph. Thereby, a play in $\mathcal{P}_\mathcal{G}$ corresponds to a play in $\mathcal{G}$ and vice versa. Note that player 0 wins all infinite plays in $\mathcal{P}_\mathcal{G}$ while the defender wins all infinite plays in $\mathcal{G}$. By construction a position (g, e) is a deadend in $\mathcal{P}_\mathcal{G}$ if and only if g is a deadend in $\mathcal{G}$ or $e = \bot$. Hence, a play is won by the defender (resp. attacker) in $\mathcal{G}$ if and only if the corresponding play is won by player 0 (resp. player 1) in $\mathcal{P}_\mathcal{G}$.

Let $e \in \mathcal{E}$ and $g \in G$. Since parity games are positionally determined, there exists a positional winning strategy s for player 0 or player 1 when starting in (g, e). Define $s_p : G^* G_p \to G$ by setting $s_p(g_0 ... g_n) := g'$ with $s\,(g_n, EL(g_0 ... g_n, e)) = (g', e')$ if $g_n \in G_p$, where $G_p = G_d$ if player 0 wins and $G_p = G_a$ else. Note that this strategy is an energy-positional winning strategy. ◀

Complexity

▶ **Theorem 28** (Complexity). *Let $\mathcal{G} = (G, G_a, \rightarrowtail)$ be a Galois energy game over $(\mathcal{E}, \leqslant)$. Further, let*

- *o be the branching degree of the underlying graph,*

- $t_{\sup}$ *be an upper bound of the time to compute the supremum of two energies,*
- $t_{\leqslant}$ *be an upper bound of the time to compute the comparison of two energies,*
- $t_{\circlearrowleft}$ *be an upper bound of the time to compute* $u^{\circlearrowleft}(e)$ *for any update* u *in* $\mathcal{G}$ *and* $e \in \mathcal{E}$*, and*
- $e_{worst} := \sup\{u_1^{\circlearrowleft} \circ ... \circ u_i^{\circlearrowleft}(\min \mathcal{E}) \mid i \in \{0, ..., |G|-1\} \ \wedge \ u_1, ..., u_i \text{ are updates in } \mathcal{G}\}$ *be the highest energy obtainable using up to* $|G|-1$ *reverse updates applied to the least element of* $\mathcal{E}$*.*

Then, Algorithm 1 on input $\mathcal{G}$ *terminates in a time in*

$$\mathcal{O}\left(|G|^2 \cdot o \cdot hgt(e_{worst}) \cdot wdh(e_{worst})^2 \cdot (t_{\circlearrowleft} + t_{sup} + wdh(e_{worst}) \cdot t_{\leqslant})\right),$$

and in $\mathcal{O}\left(|G|^2 \cdot o \cdot wdh(e_{worst})^2 \cdot (t_{\circlearrowleft} + t_{sup} + wdh(e_{worst}) \cdot t_{\leqslant})\right)$*, if* $\mathcal{G}$ *is declining.*

In both cases, the output is calculated using space in $\mathcal{O}\left(|G| \cdot wdh(e_{worst}) \cdot s_{\mathcal{E}}\right)$ *where* $s_{\mathcal{E}}$ *is an upper bound for the space needed to store any energy.*

Proof. We start our arguments by considering the number of needed iterations. If all successors of a defender position g_d were previously assigned a non-empty set, then g_d will be assigned a non-empty set in the next iteration. Similarly, if any successor of an attacker position g_a was assigned a non-empty set, then g_a will be assigned a non-empty set next. After $|G|$ iterations the information, which positions are winnable for the attacker, has traveled from defender deadlocks to all other winnable positions, i.e. $\forall g \in G.\ \mathsf{Win}_a(g) \neq \varnothing \longleftrightarrow \mathsf{Iteration}^{|G|}(\mathbf{0})(g) \neq \varnothing$. This allows us to calculate an upper bound for the energies being assigned to positions during the run of the algorithm, i.e. e_{worst}.

If $\mathcal{G}$ is declining, then cycles do not benefit the attacker and a fixed point is reached in $\mathcal{O}(|G|)$ iterations. In both, declining and non-declining games, only sets in $\mathsf{antichain}_{\leqslant e_{worst}}$ are assigned to positions as candidates for minimal attacker winning budgets. Each set in $\mathsf{antichain}_{\leqslant e_{worst}}$ can only truly be updated at most as many times as the maximal length of chains in $\mathsf{antichain}_{\leqslant e_{worst}}$ w.r.t. inclusion of upward-closures, i.e. at most $\sup_{\mathbb{N}}\{i \in \mathbb{N} \mid \exists A_1, ..., A_i \in \mathsf{antichain}_{\leqslant e_{worst}}.\ \forall j \in \{1, ..., i-1\}.\ A_j \subsetneq \uparrow A_{j+1}\} = \mathrm{hgt}(e_{worst})$ times. Thus, in non-declining energy games, the algorithm terminates after at most $1 + |G| + |G| \cdot \mathrm{hgt}(e_{worst}) \in \mathcal{O}(|G| \cdot \mathrm{hgt}(e_{worst}))$ iterations of the while-loop.

The computing time needed for operations outside the while-loop is negligible. We now focus on the time needed to execute one iteration, where line 7's time is dominated by that of line 9. The following procedure inspired by Brihaye and Goeminne [8] calculates line 9, i.e. $\mathsf{win}[\mathsf{g}]$ for a defender position g.

```
1 def compute_new_win(𝒢, old_win, g):
2 |   new := Min ℰ
3 |   for g′ with g ↣^u g′ :
4 |   |   new := Min{sup{e′, u^↺(e_{g′})} | e′ ∈ new ∧ e_{g′} ∈ old_win[g′]}
5 |   return new
```

Note that the repeated application of Min ensures that $\mathrm{wdh}(e_{worst})$ is an upper bound for the size of new as well as $\mathsf{old_win}[g']$ for all $g' \in G$. Therefore, line 4 in the procedure can be calculated in $\mathcal{O}\left(\mathrm{wdh}(e_{worst})^2 \cdot (t_{\circlearrowleft} + t_{\sup} + \mathrm{wdh}(e_{worst}) \cdot t_{\leqslant})\right)$. The for-loop in line 3 adds a factor of the branching degree o, while line 5 adds $|G|$. This yields a running time of Algorithm 1 in $\mathcal{O}\left(|G|^2 \cdot o \cdot \mathrm{hgt}(e_{worst}) \cdot \mathrm{wdh}(e_{worst})^2 \cdot (t_{\circlearrowleft} + t_{sup} + \mathrm{wdh}(e_{worst}) \cdot t_{\leqslant})\right)$ where $\mathrm{hgt}(e_{worst})$ may be omitted in the case of declining energy games.

Since each $\mathsf{win}[g]$ calculated contains at most $\mathrm{wdh}(e_{worst})$ elements, the space needed to compute the output is proportional the space of $|G| \cdot \mathrm{wdh}(e_{worst})$ energies. ◀

Open Bisimilarity for the π-Calculus with Mismatch

Tiange Liu
School of Computing, The Australian National University, Canberra, Australia

Alwen Tiu
School of Computing, The Australian National University, Canberra, Australia

Ross Horne
University of Strathclyde, Glasgow, UK

Abstract

Open bisimilarity is an equivalence relation for the π-calculus that is also congruence, making it suitable to use in compositional reasoning for mobile processes and communication protocols. The original definition of open bisimilarity, due to Sangiorgi, does not account for the mismatch operator, that is crucial in modelling real-world protocols. When mismatch is present, the congruence property no longer holds for open bisimilarity. In a LICS 2018 paper, Horne et al. proposed an extension of open bisimilarity, using a history-indexed class of relations, to address this problem. That definition, however, turns out to be non-compositional as we shall demonstrate in this paper. This paper presents a new definition of open bisimilarity in the π-calculus that incorporates mismatch. This is achieved by augmenting the transition semantics of the π-calculus with an explicit assumption about name distinctions, and by requiring that open bisimulation to be closed under an arbitary extension of the name distinctions assumption. We then prove that the resulting open bisimilarity is both an equivalence relation and a congruence.

2012 ACM Subject Classification Theory of computation → Process calculi

Keywords and phrases mismatch, open bisimilarity, pi calculus

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2025.30

1 Introduction

The π-calculus is a mathematical model developed for studying and describing concurrent systems, particularly focusing on how these systems can communicate and change their structures dynamically. It was introduced by Milner, Parrow, and Walker in the late 1980s and early 1990s [18,19]. Within this framework, equivalence relations, known as bisimilarities, were developed to compare the behaviors of processes. Milner, Parrow and Walker initially defined the late and early bisimilarity [19], which provide foundational approaches for reasoning about process equivalence. Later, Sangiorgi and Walker extended these notions by introducing open bisimilarity [22] and quasi-open bisimilarity [23], which allow for a more flexible treatment of free and bound names. These extensions address limitations in the original definitions, particularly with respect to handling name instantiation and substitution.

An important property for an equivalence relation in the π-calculus is that it is a congruence. A congruence relation ensures that if two processes are bisimilar, they remain bisimilar in any context. While late and early bisimilarities fail to satisfy congruence, as they are not preserved by the input prefix operator, both open and quasi-open bisimilarities are congruence relations. This property guarantees that if two processes are shown to be open or quasi-open bisimilar, they will exhibit identical behavior under any contextual composition.

However, Sangiorgi's definition does not account for the mismatch operator $[x \neq y]P$, which is crucial for modeling protocols that require explicit distinctions between names. The mismatch operator, denoted as $[x \neq y]P$, enables a process P to proceed only if the names x and y are distinct. If x and y are the same, the process cannot proceed and essentially behaves like the deadlock process 0.

© Tiange Liu, Alwen Tiu, and Ross Horne;
licensed under Creative Commons License CC-BY 4.0

We are particularly interested in congruence relations (open and quasi-open bisimilarity). With congruence, one can prove equivalence for individual components of a system and infer equivalence for the entire system without re-evaluating interactions in every context. This greatly simplifies reasoning about correctness and facilitates the modular design of mobile systems, where processes often operate in dynamic and evolving environments. While quasi-open bisimilarity with mismatch has been explored and formalised in previous work [13], open bisimilarity with mismatch remains an open topic. In this paper, we fill this gap by extending Sangiorgi's original definition of open bisimilarity to include mismatch.

Our interests in (quasi-)open bisimulation stem from its operational nature that brings it closer to implementation. In particular, it gets rid of one source of infinity – the quantification over all input when reasoning about processes that are parameteric on the input they receive. The lazy treatment of input values in open bisimulation is similar to that of symbolic bisimulation [7, 12], but with the additional advantage of being a congruence. The former allows for practical implementations of bisimulation checkers for the π-calculus and its extensions (see e.g., [5, 26, 27]), while the latter has been shown to be crucial in reasoning about privacy properties of security protocols formalised in the applied-pi calculus, a cryptographic extension of the π-calculus [15]. The addition of mismatch enables processes to make decisions based on name inequalities. This capability allows one to, for example, encode a typical if-then-else construct in programming languages and protocols, with the else branch corresponds to the mismatch prefix. This use of the mismatch operator is important in modeling real-world protocols, as such protocols typically incorporate various error conditions (that are typically captured in the else-branch of a decision point) that could potentially lead to privacy leakage, as shown in cryptographic extensions of the π-calculus, e.g., [4, 9]. By studying mismatch in a minimal setting in the π-calculus, we hope to gain insights into its uses in (cryptographic) extensions of the π-calculus.

In most presentations of the π-calculus including the original syntax, the mismatch operator is absent for several reasons. Firstly, introducing mismatch can violate the monotonicity property of substitutions in the π-calculus. Substitutions typically should not reduce the capability of a process for action [24, Chapter 1.1]. Formally, if $P \xrightarrow{\alpha} P'$, then we want $P\sigma \xrightarrow{\alpha\sigma} P'\sigma$ for all possible σ [20]. However, mismatch can lead to situations where two originally distinct names become identical after substitution, resulting in a process being less capable of performing actions due to name equality. For example, $[x \neq y]\tau$ can perform a τ-transition, but when we apply the substitution $\{{}^{y}/_{x}\}$, the process transforms into $[y \neq y]\tau$ which equals 0. Secondly, a naive extension of open bisimilarity with mismatch would result in a non-compositional open bisimilarity, breaking the congruence property that ensures the open bisimilarity is preserved under all contexts. We will use counterexamples to illustrate this point in the next section.

To maintain the monotonicity property, one crucial measure is to impose some restrictions on the substitution of names. We stipulate that, without any further assumptions, $[x \neq y]P$ can only execute if it is guaranteed that x and y can never become equal under an arbitrary context that can potentially modify x or y, e.g., if the process is nested inside an input prefix binding x or y. To account for the interactions between input names and restricted names (induced by scope extrusion in the π-calculus), a notion of *name distinctions* [22] is introduced in both the operational semantics of the π-calculus and our new definition of open bisimulation. Under this definition, these two processes $[x \neq y]\tau$ and τ can be distinguished using open bisimilarity because the former can perform a τ-transition only if the inequality between x and y is ensured, i.e., x and y cannot be equal under any possible substitutions, while the latter can perform the τ-transition unconditionally.

Horne et al. [13] introduced an extended definition of quasi-open bisimilarity that incorporates mismatch. In the same work, they also explored open bisimilarity with mismatch, proposing an extended definition based on a history-indexed syntax that allows histories to be prepended. An alternative extension of open bisimilarity was proposed in [17], also utilizing history-indexed syntax, but leveraging on a certain rigidisation relation. However, as we will demonstrate in Section 3, these definitions are fundamentally unsound in a strong sense, i.e., they are not preserved by parallel composition. We propose here a new definition of open bisimilarity that extends only slightly Sangiorgi's original definition, by essentially requiring open bisimilarity to be preserved by additions of name distinctions. We then prove that this new open bisimilarity is both an equivalence relation and a congruence.

Outline. In Section 2 we review related work on process calculi featuring mismatch. In Section 3, we revisit the original syntax and operational semantics of the π-calculus (excluding mismatch) and review Sangiorgi's foundational definitions of distinction and open strong bisimilarity. As all the bisimulation relations we discuss in this paper are strong bisimulation relations (where the silent transitions are taken into account in the bisimulation game), we shall simply refer to these relations as bisimulation rather than strong bisimulation. Our results should extend to weak bisimulation as well, as the issue of name distinctions is orthogonal to the strong vs weak semantics of bisimulation. In Section 4, we discuss prior attempts to incorporate mismatch into open bisimilarity and illustrate their issues through counterexamples. In Section 5, we add mismatch to the syntax and operational semantics, subsequently defining open bisimilarity with mismatch. We present a slightly revised operational semantics indexed by distinction D. Then this new definition of open bisimilarity is proven to be a equivalence relation and a congruence relation. In Section 6, we show that our open bisimilarity cannot be represented using the previously history-indexed semantics. Finally, Section 7 provides conclusions and outlines future work.

2 Related Work

We touch on some related work on process calculi featuring mismatch and work on open bisimilarity.

Parrow and Victor introduced the fusion calculus [21], which is a unified communication model where name equivalence and name fusion replace name-passing and scope extrusion, simplifying the core concepts of mobility. The Fu and Yang provided complete axiomatizations for fusion calculus both with and without the mismatch operator. The paper discusses how mismatch can complicate the semantics but is essential for certain types of behavioral equivalences and conditions. In [11], the authors investigated weak bisimulation congruences for finite π-calculus processes, focusing on challenges posed by the mismatch operator. The paper demonstrates that the standard definition of weak open congruence leads to problematic equivalence relations when mismatch is included, and proposes two alternatives: late open congruence and early open congruence.

Open bisimulation shares many similarities with symbolic bisimulation, in particular in its "lazy" treatment of name instantiations. Symbolic bisimulation has been extensively studied for different calculi: For value-passing CCS, Hennessy and Lin [12] generalised the standard notion of labelled transition graphs to symbolic transition graphs, allowing the operational semantics of value-passing processes to be represented in finite terms even when the underlying transition graph is infinite. This work introduces algorithms applicable to early and late bisimulation equivalences, extending the scope of standard bisimulation techniques.

Symbolic bisimulation for the π-calculus was addressed in [7], where symbolic transition systems are tailored to handle mobility and dynamic process communication, culminating in a sound and complete proof system for symbolic bisimulation. Delaune et al. proposed a symbolic semantics for the finite applied π-calculus [10], adapting symbolic bisimulation for modeling cryptographic protocols. While this approach is sound, it is not complete, primarily automating observational equivalence under constraints from external inputs. A more comprehensive framework was proposed in [16], which achieved both soundness and completeness for symbolic bisimulation in the full applied π-calculus. This framework also supports replication and other complex constructs, making it suitable for equivalence checking in dynamic and security-critical systems.

The original semantics provided for the applied π-calculus is sound and complete with respect to observational equivalence [1]. However, as we know, observational equivalence is not closed under input prefix, as per its definition in the context of protocol verification. This distinguishes it fundamentally from open bisimulation. Early work on symbolic bisimulation for CCS and its subsequent extension to the π-calculus can be characterised as "classical". In these approaches, labelled transition semantics are indexed by constraints – arising from name equality or inequality in the π-calculus, and more complex terms in the applied π-calculus. Different solutions to these constraints correspond to different continuations of a given symbolic process. The constraint languages employed are based on classical logic, inherently incorporating the principle of the excluded middle for names ($x = y \vee x \neq y$). As demonstrated in [3], open bisimulation, in contrast, is inherently intuitionistic. Consequently, symbolic bisimulations are distinct from open bisimulation at that fundamental level.

Horne and Mauw have explored open bisimilarity in the applied π-calculus that features mismatch. In [14], they introduced a generalisation of the distinction relation to sets of inequalities between terms, where inequalities act as constraints that exclude unifying messages under an equational theory. Their framework was developed to support a chain of techniques for verifying privacy properties in a real-world ePassport case study, highlighting the practical relevance of open bisimilarity in applied security protocols. While their work demonstrates the applicability of the concept in a more expressive calculus, it does not establish a congruence result for the proposed bisimilarity. In contrast, this paper addresses this theoretical gap in the simpler setting of the π-calculus with mismatch, where many of the core challenges of compositional reasoning arise.

3 Preliminary definitions

In this section we review the syntax and operational semantics of the finite fragment (i.e., the fragment without replication or recursion) of the π-calculus and open bisimilarity.

3.1 π calculus

We assume a countably infinite set of *names*, whose members are ranged over by lower-case letters such as x, y and z. Figure 1 shows the original syntax and operational semantics of the π-calculus. We denote with $\mathcal{P}$ the set of processes.

There are three kinds of prefixes for processes, i.e., the silent prefix, the output prefix, and the input prefix, indicating the capabilities of actions. The silent prefix represents an internal action that is not visible to the outside environment. It is used to model internal computation or synchronization that does not involve any communication with the external environment. The output prefixes represent an action where the process sends the name y along channel x. In free output $\overline{x}y$, the name y is not bound by the action, meaning it

$$\pi ::= \quad \tau \quad \text{(silent)} \qquad\qquad P ::= \quad 0 \quad \text{(deadlock)}$$
$$\overline{x}y \quad \text{(free output)} \qquad\qquad \nu x.P \quad \text{(new)}$$
$$x(y) \quad \text{(input)} \qquad\qquad \pi.P \quad \text{(action)}$$
$$[x = y]P \quad \text{(match)}$$
$$P|P \quad \text{(par)}$$
$$P + P \quad \text{(choice)}$$

$$\frac{}{\pi.P \xrightarrow{\pi} P}\ (\textsc{Act}) \qquad \frac{P \xrightarrow{\pi} Q \quad z \notin \mathrm{bn}(\pi) \cup \mathrm{fn}(\pi)}{\nu z.P \xrightarrow{\pi} \nu z.Q}\ (\textsc{Res}) \qquad \frac{P \xrightarrow{\overline{x}z} Q \quad z \neq x}{\nu z.P \xrightarrow{\overline{x}(z)} Q}\ (\textsc{Open})$$

$$\frac{P \xrightarrow{\overline{x}(z)} P' \quad Q \xrightarrow{x(z)} Q'}{P|Q \xrightarrow{\tau} \nu z.(P'|Q')}\ (\textsc{Close}) \qquad \frac{P \xrightarrow{\overline{x}y} P' \quad Q \xrightarrow{x(z)} Q'}{P|Q \xrightarrow{\tau} P'|Q'\{y/z\}}\ (\textsc{Com}) \qquad \frac{P \xrightarrow{\pi} R}{P + Q \xrightarrow{\pi} R}\ (\textsc{Sum})$$

$$\frac{P \xrightarrow{\pi} Q \quad \mathrm{bn}(\pi) \cap \mathrm{fn}(R) = \emptyset}{P|R \xrightarrow{\pi} Q|R}\ (\textsc{Par}) \qquad \frac{P \xrightarrow{\pi} Q}{[x = x]P \xrightarrow{\pi} Q}\ (\textsc{Match})$$

Figure 1 The original syntax and late transitional semantics of the π-calculus. Their symmetric variants are omitted.

is a freely chosen name. The input prefix represents an action where the process waits to receive a name on channel x, and the received name is bound within the scope of the input action. The processes can be of the form defined in Figure 1. Besides the action prefixes that we just introduced, $\nu x.P$ represents the creation of a new, private channel x within the process P. The scope of the name x is restricted to the process P, meaning x is not visible outside P. Match represents a conditional process that behaves like P can evolve if and only if x is equal to y. Parallel composition represents two processes running in parallel. Both processes can proceed independently or interact with each other through communication. Choice represents a process that can choose to behave either like process P or Q. The choice is nondeterministic, meaning the process can proceed with either option.

3.2 Distinction and open bisimilarity

As previously mentioned, names in the π-calculus are not inherently distinguished by how they can be instantiated. However, there are situations where we need to impose additional requirements on certain names, such as ensuring that some specific names remain distinct. Therefore, *distinctions* [22] are used to manage and track name inequalities explicitly. They help to maintain and enforce the conditions under which two processes can be considered behaviorally equivalent, taking into account the different ways names can be treated (bound or free) and ensuring that comparisons are meaningful even when names are abstracted or bound within processes.

A *substitution* is a mapping from names to names that is the identity everywhere except for a finite set of names, called its *domain*. Substitutions are ranged over by σ, θ, ρ, possibly with subscripts. We use the notation $\{y_1/x_1, \ldots, y_n/x_n\}$ to enumerate a substitution mapping x_i to y_i, for $i \in \{1, \ldots, n\}$. Given a substitution σ and names x and y, the substitution $\sigma[y/x]$ is defined as follows: $(\sigma[y/x])(z) = \sigma(z)$ if $z \neq x$, and $\sigma(x) = y$. Applications of a substitution to a syntactic expression (terms, processes, etc) are written in a postfix notation, e.g., $P\sigma$ denotes the result of applying σ to P. This notational convention is extended homomorphically to sets and relations. We require that the application of a substitution to an expression containing binders are capture-avoiding.

▶ **Definition 1** (Distinction). *A distinction D is a finite symmetric and irreflexive relation on names. A substitution σ* respects *D if $(x, y) \in D$ implies $x\sigma \neq y\sigma$. We denote with $\mathcal{D}$ the set of all distinctions. Given sets of names A and B, the distinction $A \otimes B$ is defined as:*

$$A \otimes B = \{(a, b), (b, a) \mid a \in A, b \in B, a \neq b\}$$

This means that $A \otimes B$ contains all ordered pairs (a, b) and (b, a) such that a is an element of A, b is an element of B, and $a \neq b$.

Sangiorgi defined the open bisimilarity as a family of equivalence relations indexed by distinctions. Given two family of relations $\mathbb{R} = \{R^D\}_{D\in\mathcal{D}}$ and $\mathbb{S} = \{S^D\}_{D\in\mathcal{D}}$, we say $\mathbb{R}$ is subsumed by $\mathbb{S}$, written $\mathbb{R} \preceq \mathbb{S}$, if for every $D \in \mathcal{D}$, we have $R^D \subseteq S^D$.

▶ **Definition 2** (Sangiorgi's open bisimilarity). *A family of relations $\mathbb{R} = \{R^D\}_{D\in\mathcal{D}}$ is an* open bisimulation *if for every $D \in \mathcal{D}$, whenever PR^DQ:*

- *For all σ respecting D, $(P\sigma)\ R^{D\sigma}\ (Q\sigma)$;*
- *If $P \xrightarrow{\alpha} P'$ and α is not a bound output, then $\exists Q'$ s.t. $Q \xrightarrow{\alpha} Q'$ and $P'R^DQ'$;*
- *If $P \xrightarrow{\bar{x}(z)} P'$ and z is fresh, then $\exists Q'$ s.t. $Q \xrightarrow{\bar{x}(z)} Q'$ and $P'R^{D'}Q'$, where $D' = D \cup (\{z\} \otimes \mathrm{fn}(P, Q))$.*

We refer to $\sim_o^D$ as open D-bisimilarity iff there exists an open bisimulation $\mathbb{R}$ and an $R^D \in \mathbb{R}$ such that PR^DQ. Open bisimilarity *refers to a special case of open D-bisimilarity where $D = \emptyset$.*

However, as we mentioned in the introduction, directly adding mismatch to Sangiorgi's open bisimilarity would break the monotonicity property. The mismatch operator typically behaves as "if $x \neq y$ then P" [20], formally,

$$\frac{P \xrightarrow{\pi} Q}{[x \neq y]P \xrightarrow{\pi} Q}\ (\textsc{Mismatch}) \tag{1}$$

This definition is obviously not closed under substitution. As we illustrated in the introduction, $[x \neq y]P$ can execute as P, but when we apply $\{y/x\}$ to the process, $[y \neq y]P$ cannot perform any transitions.

4 History-indexed formulations of bisimilarity

As mentioned in the introduction, maintaining the monotonicity property requires imposing certain restrictions on name substitution. Several works have attempted to address this challenge. In this section, we review existing approaches to incorporating mismatch into open bisimilarity and discuss the shortcomings of each approach.

Horne at al. [13] presented an extended open bisimilarity using the notion of history h, which is a list of names annotated with either i (denoting an input name) or o (denoting an output name). A history h captures the sequence of names that a process sends and receives during its transitions. It is essentially a (partial) trace of a process. The use history-indexed open bisimulation was first proposed in [25] for the spi-calculus [2], a cryptographic extension of the π-calculus, although the idea dates back to the encoding of the π-calculus in a logical framework [27], where the quantifier alternation used in statements surrounding bisimilarity captures the notion of histories we use here. A history or a quantifier alternation gives rise to an implicit encoding of distinction, and is very natural from both the logical and an implementation perspective. The latter, for example, could build on existing logical frameworks (such as [5]) to provide a simple implementation of open bisimulation. It is thus

a natural question to ask whether one could reformulate open bisimulation using histories to encode distinctions. The answer for the case without mismatch is in the affirmative; in the presence of mismatch, this turns out to be not true. We shall come back to this in Section 6.

We simplify the notion of histories from [25] to include only names, rather than general terms. Names annotated with o (written as z^o) represent the output names that a process emits in its bound output transitions. Conversely, names annotated with i (written as z^i) signify the symbolic inputs (i.e., variables) received by the process. The difference between these two types of annotations is formalised in the following definition of respectful substitutions.

▶ **Definition 3** (Substitutions with respect to a history). *A substitution* σ respects h *if, for all* h', h'' *and* x *such that* $h = h' \cdot x^o \cdot h''$*, we have* $x\sigma = x$ *and* $y\sigma \neq x$ *for all* $y \in h'$*.*

In this context, the o-annotated names act like constants, while i-annotated names act like scoped variables, with their scoping determined by their relative positions in the history. Viewing a history as a trace of names that a process inputs and outputs, this scoping ensures that a name received earlier in the trace cannot be identified with a fresh name outputted later. Tiu and Miller have discussed in [27, Corollary 22] that open bisimulation indexed by histories does not affect the resulting notion of bisimilarity compared to Sangiorgi's original definition.

Given a history h, it is easy to generate a distinction that can functionally replace h. Let P and Q be two processes, h be a history. Define a distinction D_h from h such that

$$D_h = \{(x_i, x_j) \mid i \neq j \text{ and } x_i, x_j \in h^o, \text{ or } i < j \text{ and } x_i \in h^i, x_j \in h^o\} \tag{2}$$

The converse does not hold, as not all possible distinctions can be represented as histories. However, the open bisimilarity relation (i.e., the largest open bisimulation), defined via the history-indexed style, coincides with Sangiorgi's original definition (Definition 2), as one can show that open-bisimilarity between two processes (under the empty distinction) can be witnessed by a bisimulation relation indexed only by distinctions that are representable as histories [27].

With the history-indexed semantics, the operational semantics of mismatch can be adapted as follows.

$$\frac{h : P \xrightarrow{\pi} Q \quad h \models x \neq y}{h : [x \neq y]P \xrightarrow{\pi} Q} \text{ (Mismatch)}$$

where $h \models x \neq y$ iff $x\sigma \neq y\sigma$ for all σ respecting h. Other operational semantics can be indexed by history similarly.

This semantics successfully preserves the monotonicity rule mentioned in the introduction. The history-indexed monotonicity rule can be stated as: Suppose $h : P \xrightarrow{\pi} Q$. Then $h\sigma : P\sigma \xrightarrow{\pi\sigma} Q\sigma$ for all σ that respect h and satisfy for all $x \in \text{bn}(\pi)$, $y\sigma = x$ iff $x = y$. The proof that monotonicity holds with the history-indexed mismatch rule is provided in [17].

While the monotonicity issue has been resolved, the congruence problem remains challenging. Let us look at a history-indexed open bisimilarity defined in [17].

▶ **Definition 4** (History-indexed open bisimilarity [17]). *An* open bisimulation *is a history-indexed collection* $\{\mathcal{B}_o^h \mid h \in \mathcal{H}\}$ *of symmetric relations on processes such that whenever* $P\mathcal{B}_o^h Q$*:*

- *For all substitutions* σ *respecting* h*, we have* $P\sigma\mathcal{B}_o^{h\sigma}Q\sigma$*.*
- *If* $h : P \xrightarrow{\alpha} P'$ *then* $\exists Q'$ *s.t.* $h : Q \xrightarrow{\alpha} Q'$ *and* $P'\mathcal{B}_o^h Q'$*, where* α *is of the form* τ *or* $\bar{x}y$*;*

- *If* $h : P \xrightarrow{\bar{x}(z)} P'$ *and* z *is fresh, then* $\exists Q'$ *s.t.* $h : Q \xrightarrow{\bar{x}(z)} Q'$ *and* $P' \mathcal{B}_o^{h \cdot z^o} Q'$*;*
- *If* $h : P \xrightarrow{x(z)} P'$ *and* z *is fresh, then* $\exists Q'$ *s.t.* $h : Q \xrightarrow{x(z)} Q'$ *and* $P' \mathcal{B}_o^{h \cdot z^i} Q'$*.*

The pointwise union of all open bisimulations is denoted by $\{\sim_o^h \mid h \in \mathcal{H}\}$*. We refer to* $\sim_o^h$ *as* open h-bisimilarity*. We write* $P \sim_o^{h'} Q$ *if there exists an open bisimulation* $\{\mathcal{B}_o^h \mid h \in \mathcal{H}\}$ *and a history* h' *with only* i*-annotated names such that* $P \mathcal{B}_o^{h'} Q$ *and* $\mathrm{fn}(P, Q) \subseteq h'$*. We call* P *and* Q open bisimilar*.*

This definition aligns with Sangiorgi's open bisimilarity (Def. 2). In the third clause with bound output transitions, the history (environment) is updated with a fresh private name, which serves the same function as adding the fresh name to the distinction set.

This definition is not congruent in the presence of the mismatch operator. The process $[x \neq y]\tau$ under the history $h = x^i \cdot y^i$ is open bisimilar to 0, as there exists a respectful substitution $\{x/y\}$ that invalidates $x \neq y$, thereby preventing the τ-transition from $[x \neq y]\tau$. However, this open bisimilarity does not extend to $\bar{a}(x).[x \neq y]\tau$ and $\bar{a}(x).0$ under the history $h' = y^i \cdot x^o$, since $[x \neq y]$ always holds under the history h'.

To address this limitation, a few studies have been conducted. Horne et. al. [13] provided an extended version of the history-indexed open bisimilarity that allows to extend histories in the past, which gives rise to a strictly coarser semantics. Their definition can be stated as follows.

▶ **Definition 5** (Extended open bisimilarity [13])**.** *A symmetric relation* R *indexed by an environment is an open bisimulation whenever, if* $P R^h Q$ *the following hold:*

- *For all* σ *respecting* h*,* $P\sigma R^{h\sigma} Q\sigma$*;*
- *For any history* h'*, we have* $P R^{h' \cdot h} Q$*;*
- *If* $h : P \xrightarrow{\alpha} P'$ *and* α *is of the form* τ *or* $\bar{x}y$*, then* $\exists Q'$ *s.t.* $h : Q \xrightarrow{\alpha} Q'$ *and* $P' R^h Q'$*;*
- *If* $h : P \xrightarrow{\bar{x}(z)} P'$ *and* z *is fresh, then* $\exists Q'$ *s.t.* $h : Q \xrightarrow{\bar{x}(z)} Q'$ *and* $P' R^{h \cdot x^o} Q'$*.*
- *If* $h : P \xrightarrow{x(z)} P'$ *and* z *is fresh, then* $\exists Q'$ *s.t.* $h : Q \xrightarrow{x(z)} Q'$ *and* $P' R^{h \cdot x^i} Q'$*.*

Open bisimilarity $\sim_o$ *is defined such that* $P \sim_o Q$ *holds whenever there exists an open bisimulation* R *such that* $P R^{x_1^i \cdot x_2^i \cdot \ldots \cdot x_n^i} Q$ *holds, where* $\mathrm{fn}(P) \cup \mathrm{fn}(Q) \subseteq \{x_1, x_2, \ldots, x_n\}$*.*

The above definition extends the original open bisimilarity by the second clause allowing for the prepending of history, enabling the incorporation of additional environmental information.

As it turns out, however, this semantics is not compositional.

▶ **Example 6.** We look at the following two processes:

$$P = \nu k, \ell.\bar{a}k.\bar{a}\ell.a(z).([k = z]\tau + [k \neq z]\tau) \quad \text{v.s.} \quad \nu k, \ell.\bar{a}k.\bar{a}\ell.a(z).([\ell = z]\tau + [\ell \neq z]\tau) \ = Q$$

Under Definition 5, P and Q are open bisimilar (so we are treating a as an i-annotated name). After three transitions, we compare

$$P' = \ [k = z]\tau + [k \neq z]\tau \quad \text{v.s.} \quad [\ell = z]\tau + [\ell \neq z]\tau \ = Q'$$

under the history $a^i \cdot k^o \cdot \ell^o \cdot z^i$ We cannot distinguish the two processes because the history does not indicate the equality either between k and z or between ℓ and z. However the above processes can be distinguished when placed in the following context.

$$a(w) \mid [\cdot]$$

A possible partial trace for $a(w) \mid P$ is as follows:

$$\begin{aligned} & a(w) \mid P \xrightarrow{\bar{a}(k)} a(w) \mid \nu\ell.\bar{a}\ell.a(z).([k = z]\tau + [k \neq z]\tau) \\ & \xrightarrow{a(w)} \nu\ell.\bar{a}\ell.a(z).([k = z]\tau + [k \neq z]\tau) \\ & \xrightarrow{\bar{a}(\ell)} a(z).([k = z]\tau + [k \neq z]\tau) \\ & \xrightarrow{a(z)} ([k = z]\tau + [k \neq z]\tau) \end{aligned}$$

These transitions would have to be matched by $a(w) \mid Q$ for the two processes to be bisimilar, so proving their bisimilarity reduces to proving bisimilarity of P' and Q' under the history $h = a^i.k^o \cdot w^i \cdot \ell^o \cdot z^i$. So let us suppose that P' and Q' are bisimilar under h. Since bisimilarity must be closed under respectful substitution, and since $\{w/z\}$ respects h, it follows that

$$P'\{w/z\} = [k = w]\tau + [k \neq w]\tau \quad \text{must be bisimilar to} \quad Q'\{w/z\} = [\ell = w]\tau + [\ell \neq w]\tau$$

under $h' = h\{w/z\} = a^i \cdot k^o \cdot w^i \cdot l^o$. According to the definition of respectful substitutions in Definition 3, $\ell \neq w$ holds, yet neither $k = w$ nor $k \neq w$ hold. Thus $Q'\{w/z\}$ can perform a τ transition while $P'\{w/z\}$ cannot, contradicting their supposed bisimilarity.

In another attempt to address the issue [17], the authors resolved the monotonicity problem leveraging history-indexed operational semantics. Then to address the congruence property, they introduced the concept of rigidisation to refine open bisimilarity. The rigidisation of names allows us transforming an i-annotated name into an o-annotated name.

▶ **Definition 7** (Rigidisation relation). *The relation $\subseteq_{ro}$ is the smallest relation on histories such that:*

- $h \subseteq_{ro} h'$ *iff* $h = h_1 \cdot x^i \cdot h_2$ *and* $h' = h_1 \cdot x^o \cdot h_2$.
- $h \subseteq_{ro} h$.
- $\subseteq_{ro}$ *is transitively closed.*

Then open bisimilarity is extended by adding the second clause using the rigidisarion relation.

▶ **Definition 8** (Rigid open bisimilarity). *An* open bisimulation *is a history-indexed collection $\{\mathcal{B}_o^h \mid h \in \mathcal{H}\}$ of symmetric relations on processes such that whenever $P\mathcal{B}_o^h Q$:*

- *For all substitutions σ respecting h, we have $P\sigma\mathcal{B}_o^{h\sigma}Q\sigma$.*
- *For any $h' \supseteq_{ro} h$, we have $P\mathcal{B}_o^{h'}Q$.*
- *If $h : P \xrightarrow{\alpha} P'$ then $\exists Q'$ s.t. $h : Q \xrightarrow{\alpha} Q'$ and $P'\mathcal{B}_o^h Q'$, where α is of the form τ or $\bar{x}y$;*
- *If $h : P \xrightarrow{\bar{x}(z)} P'$ and z is fresh, then $\exists Q'$ s.t. $h : Q \xrightarrow{\bar{x}(z)} Q'$ and $P'\mathcal{B}_o^{h\cdot z^o}Q'$;*
- *If $h : P \xrightarrow{x(z)} P'$ and z is fresh, then $\exists Q'$ s.t. $h : Q \xrightarrow{x(z)} Q'$ and $P'\mathcal{B}_o^{h\cdot z^i}Q'$.*

The pointwise union of all open bisimulations is denoted by $\{\sim_o^h \mid h \in \mathcal{H}\}$. We refer to $\sim_o^h$ as open h-bisimilarity. *We write $P \sim_o^h Q$ if there exists an open bisimulation $\{\mathcal{B}_o^h \mid h \in \mathcal{H}\}$ and a history h with only i-annotated names such that $P\mathcal{B}_o^h Q$ and $\mathrm{fn}(P, Q) \subseteq h$. We call P and Q* open bisimilar.

This definition is also not compositional, as the following counterexample shows.

▶ **Example 9.** Consider the following processes:

$$P = a(z).a(y).a(x).([x \neq y]\tau.([x = z]\tau + [x \neq z]\tau)) \quad \text{v.s.} \quad a(z).a(y).a(x).([x \neq y]\tau.\tau) = Q$$

These two processes are rigid open bisimilar according to Definition 8. To see this, observe after three inputs we get to

$$([x \neq y]\tau.([x = z]\tau + [x \neq z]\tau) \sim_o^{z^i \cdot y^i \cdot x^i} [x \neq y]\tau.\tau$$

There are two minimal ways to pass that guard, using the rigidisation. One approach is to directly rigidise x^i to x^o, which enables the condition $[x \neq z]$. Alternatively, we can first set $x = z$, and then rigidise x, in which case $[x = z]$ is enabled. However, when these processes are placed in the following context, they are no longer rigid open bisimilar.

$$C[\cdot] = \nu y.b(x).\bar{b}(y).b(z).\nu a.(\bar{a}(z).\bar{a}(y).\bar{a}(x)|[\cdot])$$

Placing the above processes in the given context and executing for 6 steps we get:

$$[x \neq y]\tau.([x = z]\tau + [x \neq z]\tau) \sim_o^{b^i.x^i.y^o.z^i} [x \neq y]\tau.\tau$$

Now, since x appears before y is output in the history, we have $x \neq y$, so we reach the following after another τ-transition.

$$[x = z]\tau + [x \neq z]\tau \sim_o^{b^i.x^i.y^o.z^i} \tau$$

Trivially process τ on the right hand side can make a τ-transition, however, since neither $x = z$ nor $x \neq z$ are decided yet in this history, that τ-transition cannot be matched on the left hand side. This gives us a distinguishing strategy.

Therefore we violate the fundamental property of open bisimilarity – that it is a congruence. That is we have $P \sim_o^h Q$, but not $C[P] \sim_o^h C[Q]$.

5 A compositional open bisimilarity with mismatch

We now present our approach to incorporating mismatch in open bisimulation in a way that ensures the resulting bisimilarity is a congruence.

5.1 Syntax and operational semantics

This section introduces a variant of the π-calculus that incorporates mismatch. The labelled operational semantics are indexed by distinction D to address mismatch in open terms, where names are treated as variables instead of distinct constants. The syntax of processes is as in Figure 1, extended with the mismatch operator $[x \neq y]P$.

Intuitively, the mismatch operator $[x \neq y]P$ is interpreted as saying that that process behaves like P if "x and y can never be equal", that is, x and y can be proven distinct. This interpretation obviously depends on the context of name distinction under which this process is executed, so the transition judgment in our operational semantics includes an explicit reference to a distinction. The rest of the operators are interpreted in the same way as introduced in Section 3.1.

In the context of open bisimulation, names within a process can be substituted throughout execution. Consequently, the operational semantics for mismatch must account for all possible instantiations. To address this, we present the operational semantics of the π-calculus with mismatch in Figure 2. Our formulation introduces a slight variation from the standard late operational semantics by incorporating a distinction index for each transition. This index is crucial for including the mismatch operator. Another key point to observe is that the distinctions are extended with a fresh name in the Open and Res rules when the ν-binder appears in process expressions. This notation ensures that such names remain unchanged by respectful substitutions. This is crucial for dealing with mismatch when a variable is bound by a ν-binder, e.g., to conclude that $\nu x.[x \neq y]\tau$ can perform a τ-transition:

$$\dfrac{\dfrac{\dfrac{}{\{(x,y),(y,x)\} : \tau \xrightarrow{\tau} 0}\ (\textsc{Act})}{\{(x,y),(y,x)\} : [x \neq y]\tau \xrightarrow{\tau} 0}\ (\textsc{Mismatch})}{\emptyset : \nu x.[x \neq y]\tau \xrightarrow{\tau} 0}\ (\textsc{Res})$$

$$\frac{}{D : \pi.P \xrightarrow{\pi} P}\ (\textsc{Act}) \qquad \frac{D' : P \xrightarrow{\pi} Q \quad z \notin n(\pi) \quad D' = D \cup (\{z\} \otimes \mathrm{fn}(\nu z.P))}{D : \nu z.P \xrightarrow{\pi} \nu z.Q}\ (\textsc{Res})$$

$$\frac{D' : P \xrightarrow{\bar{x}z} Q \quad z \neq x \quad D' = D \cup (\{z\} \otimes \mathrm{fn}(\nu z.P))}{D : \nu z.P \xrightarrow{\bar{x}(z)} Q}\ (\textsc{Open}) \qquad \frac{D : P \xrightarrow{\bar{x}(z)} P' \quad D : Q \xrightarrow{x(z)} Q'}{D : P|Q \xrightarrow{\tau} \nu z.(P'|Q')}\ (\textsc{Close})$$

$$\frac{D : P \xrightarrow{\bar{x}y} P' \quad D : Q \xrightarrow{x(z)} Q'}{D : P|Q \xrightarrow{\tau} P'|Q'\{y/z\}}\ (\textsc{Com}) \qquad \frac{D : P \xrightarrow{\pi} Q \quad \mathrm{bn}(\pi) \cap \mathrm{fn}(R) = \emptyset}{D : P|R \xrightarrow{\pi} Q|R}\ (\textsc{Par})$$

$$\frac{D : P \xrightarrow{\pi} R}{D : P + Q \xrightarrow{\pi} R}\ (\textsc{Sum}) \qquad \frac{D : P \xrightarrow{\pi} Q}{D : [x = x]P \xrightarrow{\pi} Q}\ (\textsc{Match}) \qquad \frac{D : P \xrightarrow{\pi} Q \quad (x, y) \in D}{D : [x \neq y]P \xrightarrow{\pi} Q}\ (\textsc{Mismatch})$$

Figure 2 The late transition semantics with mismatch of the π-calculus indexed by distinction D. Their symmetric variants are omitted. For any transition $D : P \to Q$ that z is a bound name in P, we require z to be fresh from D.

▶ **Lemma 10.** *Let D be a distinction and suppose σ, θ are substitutions such that σ respects D. Then θ respects $D\sigma$ if and only if $\sigma \cdot \theta$ respects D.*

The following monotonicity property of processes ensures that any substitution does not diminish the process's capability for action. We show that our extended operational semantics with mismatch preserves the monotonicity property.

▶ **Lemma 11** (Monotonicity). *Suppose $D : P \xrightarrow{\pi} Q$. Then $D\sigma : P\sigma \xrightarrow{\pi\sigma} Q\sigma$ for all σ that respect D and satisfy for all $x \in \mathrm{bn}(\pi)$, $y\sigma = x$ iff $x = y$.*

Proof. We prove the lemma by induction on the transition rules. We show here an inductive case involving the Mismatch rule.

$$\frac{D : P \xrightarrow{\pi} Q \quad (x, y) \in D}{D : [x \neq y]P \xrightarrow{\pi} Q}.$$

We want to prove that if $D : [x \neq y]P \xrightarrow{\pi} Q$ then $D\sigma : [x\sigma \neq y\sigma]P\sigma \xrightarrow{\pi\sigma} Q\sigma$ for all σ respecting D. From the Mismatch rule we have $D : P \xrightarrow{\pi} Q$ and $(x, y) \in D$. By induction hypothesis, $D\sigma : P\sigma \xrightarrow{\pi\sigma} Q\sigma$ for all σ respecting D. Also from Lemma 10, $(x\sigma, y\sigma) \in D\sigma$. Then following the Mismatch rule, we have

$$\frac{D\sigma : P\sigma \xrightarrow{\pi\sigma} Q\sigma \quad (x\sigma, y\sigma) \in D\sigma}{D\sigma : [x\sigma \neq y\sigma]P\sigma \xrightarrow{\pi\sigma} Q\sigma}$$

◀

5.2 Open bisimilarity with mismatch

Indexing the transition rule by the distinction D resolves the monotonicity problem. However, this does not address all the issues. The congruence property of open bisimilarity is still not guaranteed. We look at a simple example.

▶ **Example 12.** Consider the following two processes: $P = [x \neq y]\tau$ and $Q = 0$. They are bisimilar ($P \sim_o^D Q$) under Sangiorgi's definition (Def. 2), because x and y are free names so that $D = \emptyset$. Yet if we place the above processes in the context $\bar{a}(x).\{\cdot\}$,

$$\bar{a}(x).([x \neq y]\tau) \not\sim_o^D \bar{a}(x).0 \qquad \text{where } D = \{(x, y), (y, x), (x, a), (a, x)\}.$$

Then after the first bound output transition, the resulting processes are no longer bisimilar according to the same definition. The former process can make a τ-transition because $(x, y) \in D$, which implies that x and y cannot be equal under all respectful substitutions. In contrast, the latter process cannot make such a transition.

We then need to revise the definition of open bisimilarity to address the congruence problem. The below definition slightly extends Definition 2 by adding the second clause, imposing the closure of open bisimulation under arbitrary extensions of distinctions. For clarity and convenience in subsequent proofs, for the rest of the paper, we denote open bisimilarity as a ternary relation, relating triplets of two processes and a distinction.

▶ **Definition 13** (Open bisimilarity). *An open bisimulation is a ternary relation $R \subseteq \mathcal{P} \times \mathcal{P} \times \mathcal{D}$ such that R is* symmetric *(i.e., $(P, Q, D) \in R$ iff $(Q, P, D) \in R$) and for all $(P, Q, D) \in R$:*

- *For all σ respecting D, $(P\sigma, Q\sigma, D\sigma) \in R$*
- *For all $D' \supseteq D$, $(P, Q, D') \in R$*
- *If $D : P \xrightarrow{\alpha} P'$ and α is not a bound output, then $\exists Q'$ such that $D : Q \xrightarrow{\alpha} Q'$ and $(P', Q', D) \in R$*
- *If $D : P \xrightarrow{\bar{x}(z)} P'$ and z is fresh, then $\exists Q'$ such that $D : Q \xrightarrow{\bar{x}(z)} Q'$ and $(P', Q', D') \in R$, where $D' = D \cup (\{z\} \otimes \mathrm{fn}(P, Q))$*

Open D-bisimilarity $\sim^D$ is such that $P \sim^D Q$ whenever there exists an open bisimulation R such that $(P, Q, D) \in R$. We write $P \sim Q$ to denote $P \sim^{\emptyset} Q$, i.e., they are open bisimilar under the empty distinction.

Consider again Example 12. We have shown that $[x \neq y]\tau \sim_o^D 0$, for $D = \emptyset$, under Definition 2. However, according to the second clause in our new definition, we can add $\{(x, y), (y, x)\}$ to D and then $D : [x \neq y]\tau \xrightarrow{\tau} 0$ while 0 cannot make such transition. Therefore $[x \neq y]\tau$ and 0 are not open bisimilar by our extended definition.

Similarly, we look at Example 6:

$$P = \nu k, \ell.\bar{a}k.\bar{a}\ell.a(z).([k = z]\tau + [k \neq z]\tau) \quad \text{v.s.} \quad \nu k, \ell.\bar{a}k.\bar{a}\ell.a(z).([\ell = z]\tau + [\ell \neq z]\tau) \ = Q$$

We have shown that P and Q are open bisimilar under the history-indexed open bisimilarity (Def. 5). The history is $k^o \cdot \ell^o \cdot z^i$, which corresponds to a distinction $D = \{(k, \ell), (\ell, k)\}$. By our new definition, however, we could define $D' \supseteq D$ such that $D' = \{(k, \ell), (\ell, k), (\ell, z), (z, \ell)\}$.

The names ℓ and z can never be equal under distinction D' while neither $k = z$ nor $k \neq z$ is decided yet. Thus there is a distinguishing strategy where the latter process plays a τ but the former cannot match the transition. Hence, under our new definition, P and Q are not open bisimilar, which avoids the non-congruence problem.

Problems with Example 9 can be resolved in a similar manner:

$$P = a(z).a(y).a(x).([x \neq y]\tau.([x = z]\tau + [x \neq z]\tau)) \quad \text{v.s.} \quad Q = a(z).a(y).a(x).([x \neq y]\tau.\tau)$$

Initially, the distinction $D = \emptyset$, as there are no bound-output names in the processes. Under our new Definition 13, if we extend D to $D' = \{(x, y), (y, x)\}$, these processes are no longer open bisimilar. This is because $x \neq y$ is guaranteed, while the equality between x and z remains undecided. This extension avoids the non-congruence issue.

A well-formulated definition of open bisimilarity should be an equivalence relation, satisfying the properties of reflexivity, symmetry, and transitivity.

▶ **Theorem 14.** *The open D-bisimilarity is an equivalence relation.*

- *Reflexivity: For all P and D, $P \sim^D P$.*
- *Symmetry: For all P, Q, D, if $P \sim^D Q$, then $Q \sim^D P$.*
- *Transitivity: For all P, Q, V and D, if $P \sim^D Q$ and $Q \sim^D V$, then $P \sim^D V$.*

Proof. The proof is provided in Appendix 8.1. ◀

To prove that our open bisimilarity (Def. 13) is a congruence, we must demonstrate that if two processes are open bisimilar, they remain open bisimilar under all possible contexts.

First, since the behavior of distinctions within contexts depends on the structure of the context, we define a formal mechanism to update a distinction D when it is placed within a context $C[\cdot]$. Specifically, if the context includes a restriction operator νz, which binds the name z, the distinction is updated by removing all references to z from D (as z becomes local to the context). For all other contexts, the distinction D remains unchanged.

This mechanism ensures that distinctions accurately reflect the constraints on free names in the process, while adhering to the scoping rules imposed by restrictions.

Given a distinction D and a name z, we denote with $D \setminus z$ the distinction obtained from D by removing all pairs of names in D containing z, i.e., $D \setminus z = \{(x, y) \mid x \neq z \text{ and } y \neq z\}$.

▶ **Definition 15** (Distinction update under context)**.** *Let $C[\cdot]$ be an arbitrary context, and $D \in \mathcal{D}$ be a distinction. We define the distinction $C[D]$ as follows:*

$$C[D] = \begin{cases} C'[D] \setminus z & \text{if } C[\cdot] = \nu z.C'[\cdot] \\ D & \text{otherwise} \end{cases}$$

Then the congruence of open bisimilarity can be formally stated as follows:

▶ **Theorem 16.** *Open bisimilarity is a D-congruence. Specifically:*

- *If $P \sim^D Q$, then $C[P] \sim^{C[D]} C[Q]$ for any context C.*

The core goal of the proof is to show that the relation $(C[P], C[Q], C[D])$ is included in bisimilarity. The most direct and intuitive approach would be to construct a bisimulation that relates them explicitly and verify that it satisfies the required properties of a bisimulation. However, finding such a bisimulation relation directly can be challenging and computationally expensive, especially when dealing with complex contexts, distinctions, and substitutions.

To address this, we adapt the *up-to technique*, a powerful method introduced by Milner and later developed extensively by Sangiorgi and others [24, Chapter 2.3], which simplifies the task of proving bisimilarity by reducing the complexity of direct comparisons. Instead of checking whether P and Q are directly bisimilar, this technique defines a function F, called a *safe function*, which transforms or approximates the bisimilarity relation. By verifying that F is *safe* – that is, F behaves consistently under various transformations such as substitutions, context embeddings, and name restrictions – the proof can be carried out more abstractly and modularly.

The key idea is that instead of requiring a bisimulation to explicitly relate every process pair at every step, we demonstrate that the function F preserves the necessary properties of bisimulation when applied to the transformed processes. In this way, the proof burden shifts from directly constructing and verifying a bisimulation to ensuring that F satisfies certain progression and closure properties. This abstraction makes the up-to technique more flexible and easier to apply in practice.

Before delving into the proof, we first define *strong open progression*, which is the foundation for defining the safe function F.

▶ **Definition 17** (Strong open progression). *Let R and S be ternary relations such that $R, S \subseteq \mathcal{P} \times \mathcal{P} \times \mathcal{D}$. We say R* strongly open progresses to *S, written $R \rightsquigarrow S$, if whenever $(P, Q, D) \in R$:*

- *For all σ respecting D, $(P\sigma, Q\sigma, D\sigma) \in S$*
- *For all $D' \supseteq D$, $(P, Q, D') \in S$*
- *If $D : P \xrightarrow{\alpha} P'$ and α is not a bound output, then $\exists Q'$ such that $D : Q \xrightarrow{\alpha} Q'$ and $(P', Q', D) \in S$*
- *If $D : P \xrightarrow{\bar{x}(z)} P'$ and z is fresh, then $\exists Q'$ such that $D : Q \xrightarrow{\bar{x}(z)} Q'$ and $(P', Q', D') \in S$, where $D' = D \cup (\{z\} \otimes \mathrm{fn}(P, Q))$*

Now we can introduce the safe function. The safe function F operates on relations and preserves the progression structure, ensuring that the bisimilarity relation is not broken during transformations. This property is formalised by the notion of *strong safety*, which guarantees that F behaves consistently with respect to progression.

▶ **Definition 18** (Strongly safe function). *A function F is strongly safe if $R \subseteq S$ and $R \rightsquigarrow S$ implies $F(R) \subseteq F(S)$ and $F(R) \rightsquigarrow F(S)$.*

We now formally define a strongly safe function and present two key lemmas that describe its properties. These can be proved similarly to the analogous lemmas in [24, Chapter 2.3].

▶ **Lemma 19.** *If F is strongly safe and $R \rightsquigarrow F(R)$, then R and $F(R)$ are included in $\sim$.*

▶ **Lemma 20.** *If F is strongly safe and $\sim \subseteq F(\sim)$, then $F(\sim) = \sim$.*

To apply the up-to technique, we define the function F that maps a relation R to a new relation capturing all possible transformations of R under arbitrary contexts C. The following lemma establishes that the function F is *strongly safe*.

▶ **Lemma 21.** *Let F be the function*

$$F(R) = \{(C[P], C[Q], C[D]) \mid \ C \text{ is an arbitrary context and } (P, Q, D) \in R\}.$$

Then F is strongly safe.

Proof. Suppose $R \subseteq S$ and $R \rightsquigarrow S$. We need to prove that $F(R) \subseteq F(S)$ and $F(R) \rightsquigarrow F(S)$. The inclusion $F(R) \subseteq F(S)$ follows directly from $R \subseteq S$ and the definition of F.

To prove the progression, we need to show that if C is an arbitrary context and $(P, Q, D) \in R$, $(C[P], C[Q], C[D]) \in F(R)$ by definition of F, then the progression conditions in Def. 17 are satisfied. We need to prove each of the claims below.

▷ Claim 1. $(C[P]\sigma, C[Q]\sigma, C[D]\sigma) \in F(S)$ for each σ that respects $C[D]$.

▷ Claim 2. $(C[P], C[Q], D_S) \in F(S)$ for all $D_S \supseteq C[D]$.

▷ Claim 3. If $C[D] : C[P] \xrightarrow{\alpha} P'$ and α is not a bound output, then $C[D] : C[Q] \xrightarrow{\alpha} Q'$ with $(P', Q', C[D]) \in F(S)$.

▷ Claim 4. If $C[D] : C[P] \xrightarrow{\bar{x}(z)} P'$ and z is fresh, then $C[D] : C[Q] \xrightarrow{\alpha} Q'$ with $(P', Q', C[D]') \in F(S)$ where $C[D]' = C[D] \cup (\{z\} \otimes \mathrm{fn}(C[P], C[Q]))$.

We present the proof of Claim 2 here, as it involves the key clause introduced in our extended definition of open bisimilarity. The remaining three claims are proved by induction on C, which requires a thorough and exhaustive case analysis. Complete proofs are provided in Appendix 8.2.

There are two cases to consider for Claim 2. In the first case, we have that the context $C[\cdot]$ does not contain any ν-binders binding a name in D_S, so we have $C[D_S] = D_S$. Let $D_1 = D \cup D_S$. Then $D_1 \supseteq D_S$ and $C[D_1] = D_S$. From the assumption that $R \rightsquigarrow S$, we have $(P, Q, D_1) \in S$, therefore $(C[P], C[Q], C[D_1]) = (C[P], C[Q], D_S) \in F(S)$ by the definition of F. For the second case, suppose that the context $C[.]$ contains a ν binder that binds a name in D_S. We consider the case where $C[.]$ binds exactly one name z in D_S (but this argument can be generalised to any number of binders). Let D_S^w be obtained from D_S by replacing z with a fresh name w. Since z is a binder in $C[.]$ it follows that $C[D]$ contains no occurrences of z. So we also have $D_S^w \supseteq C[D]$. Then by the construction in the first case, we have $(C[P], C[Q], D_S^w) \in F(S)$. Now by Claim 1, we can apply the substitution $\{z/w\}$, which respects D_S^w, to obtain $(C[P], C[Q], D_S) \in F(S)$ as required. ◀

Proof of Theorem 16. Let R be an open bisimulation and $F(R)$ defined as in Lemma 21. Clearly $R = F(R)$ when $C[\cdot]$ is empty. Therefore $R \subseteq F(R)$. Then $F(R)$ is also an open bisimulation by Lemma 20. ◀

6 Alternative definitions of open bisimilarity with mismatch

We reflect here on semantically different definitions of open bisimilarity with mismatch. A question is whether open bisimilarity, as in Def. 13, has an equivalent definition in terms of histories, by extending Def. 5 to allow more histories to be induced. In particular, an extended definition could permit more permutations such that inputs behave like universal quantifiers and outputs of fresh names like nominal quantifiers. This is a natural question, since several deep embeddings of the π-calculus work in this way.

It turns out that using (linear) histories would result in a distinct semantics. Consider the following example process.

$$P = [x \neq y][w \neq z]([x \neq z]\tau + [y \neq z]\tau + [x \neq w]\tau + [y \neq w]\tau) \quad \text{v.s.} \quad Q = [x \neq y][w \neq z]\tau$$

Clearly, according to Def. 13 these processes are not open bisimilar, since we can extend the set of distinction with the inequalities $x \neq y$ and $w \neq z$ without having to decide whether any of the other inequalities hold.

If however a ground output in a history were used to induce $x \neq y$ and $w \neq z$ then we require a history with x^i before y^o or x^o before y^i. In the same history we also require w^i before z^o or z^o before w^i. Without loss of generality suppose that x^i is before y^o and w^i is before z^o in a history. Now, since a history is an ordered sequence of inputs and outputs we have that w^i is before y^o or y^o is before w^i, as considered here.

- If w^i is before y^o, then we have that $w \neq y$ holds and hence a τ transition is enabled for both processes.
- If y^o is before w^i, then since we assumed also that x^i is before y^o and w^i is before z^o we have that x^i is before z^o. Hence $x \neq z$ holds and hence a τ transition is enabled for both processes.

Thus, regardless of how the definition is constructed, if a definition of open bisimilarity relies on a linear history to force distinctions, the processes P and Q above will be open bisimilar.

7 Conclusion and future work

This paper presents a novel definition of open bisimilarity for the π-calculus that integrates the mismatch operator. Mismatch, while crucial for modeling protocols requiring explicit name distinctions, introduces challenges such as violating the congruence property and

breaking monotonicity. Building on Sangiorgi's original framework, we address these issues by introducing a distinction-indexed operational semantics that allows for arbitrary extensions of distinctions. Our approach ensures that open bisimilarity remains a congruence relation even in the presence of mismatch. While prior approaches that relied on history-indexed relations or rigidisation strategies did not achieve compositionality, our method successfully addresses this limitation while also providing greater simplicity and conceptual clarity. Through formal proofs, we demonstrated that the extended open bisimilarity is both equivalent and congruent. This makes it a robust and expressive framework for reasoning about mobile processes and communication protocols, enabling better modeling of real-world systems.

As immediate future work, we plan to formalise the results of this paper in the theorem prover Isabelle/Nominal [28], building on an existing formalisation of the π-calculus [6].

Another avenue for possible future work is to generalise our transition semantics to be parameterised by *logical constraints*. Logically speaking, both distinctions and substitutions can be seen as expressing a conjunction of (in)equalities between names. This can be generalised in at least two possible directions: one in which we allow expressions of (in)equalities between (algebraic) terms (like those arising from the transition semantics for the applied π-calculus [1, 14]) and another where we allow other logical connectives (e.g., disjunction, implication, quantifiers). In such an extension, it is then natural to ask how such a logical constraint should be interpreted semantically, e.g., whether it should be viewed as a formula in classical (first-order) logic, or intuitionistic logic. We conjecture that the first interpretation (in classical logic) would give rise to the notion of symbolic bisimulation [12], and the second interpretation would give rise to open bisimulation. Work on symbolic bisimilarity for the ψ-calculus [8] takes a step in this direction. The ψ-calculus is parametrised on a constraint system that can be very general. Notably in Sec. 7 of the aforementioned paper a constraint logic is defined that must be classical for the completeness results in that paper to hold. By simply working without the law of excluded middle, we hypothesise that a notion of open bisimulation for ψ-calculi is obtained where the constraint logic can otherwise be general.

References

1 Martín Abadi, Bruno Blanchet, and Cédric Fournet. The applied pi calculus: Mobile values, new names, and secure communication. *J. ACM*, 65(1):1:1–1:41, 2018. doi:10.1145/3127586.

2 Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. In Richard Graveman, Philippe A. Janson, Clifford Neuman, and Li Gong, editors, *CCS '97, Proceedings of the 4th ACM Conference on Computer and Communications Security, Zurich, Switzerland, April 1-4, 1997*, pages 36–47. ACM, 1997. doi:10.1145/266420.266432.

3 Ki Yung Ahn, Ross Horne, and Alwen Tiu. A characterisation of open bisimilarity using an intuitionistic modal logic. In Roland Meyer and Uwe Nestmann, editors, *28th International Conference on Concurrency Theory, CONCUR 2017, September 5-8, 2017, Berlin, Germany*, volume 85 of *LIPIcs*, pages 7:1–7:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.CONCUR.2017.7.

4 Myrto Arapinis, Tom Chothia, Eike Ritter, and Mark Ryan. Analysing unlinkability and anonymity using the applied pi calculus. In *Proceedings of the 23rd IEEE Computer Security Foundations Symposium, CSF 2010, Edinburgh, United Kingdom, July 17-19, 2010*, pages 107–121. IEEE Computer Society, 2010. doi:10.1109/CSF.2010.15.

5 David Baelde, Andrew Gacek, Dale Miller, Gopalan Nadathur, and Alwen Tiu. The bedwyr system for model checking over syntactic expressions. In Frank Pfenning, editor, *Automated Deduction - CADE-21, 21st International Conference on Automated Deduction, Bremen, Germany, July 17-20, 2007, Proceedings*, volume 4603 of *Lecture Notes in Computer Science*, pages 391–397. Springer, 2007. doi:10.1007/978-3-540-73595-3_28.

6 Jesper Bengtson. The pi-calculus in nominal logic. *Arch. Formal Proofs*, 2012, 2012. URL: https://www.isa-afp.org/entries/Pi_Calculus.shtml.

7 Michele Boreale and Rocco De Nicola. A symbolic semantics for the pi-calculus. *Inf. Comput.*, 126(1):34–52, 1996. doi:10.1006/INCO.1996.0032.

8 Johannes Borgström, Ramunas Gutkovas, Ioana Rodhe, and Björn Victor. The psi-calculi workbench: A generic tool for applied process calculi. *ACM Trans. Embed. Comput. Syst.*, 14(1):9:1–9:25, 2015. doi:10.1145/2682570.

9 Vincent Cheval, Hubert Comon-Lundh, and Stéphanie Delaune. Trace equivalence decision: negative tests and non-determinism. In Yan Chen, George Danezis, and Vitaly Shmatikov, editors, *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, pages 321–330. ACM, 2011. doi:10.1145/2046707.2046744.

10 Stéphanie Delaune, Steve Kremer, and Mark Ryan. Symbolic bisimulation for the applied pi calculus. In Vikraman Arvind and Sanjiva Prasad, editors, *FSTTCS 2007: Foundations of Software Technology and Theoretical Computer Science, 27th International Conference, New Delhi, India, December 12-14, 2007, Proceedings*, volume 4855 of *Lecture Notes in Computer Science*, pages 133–145. Springer, 2007. doi:10.1007/978-3-540-77050-3_11.

11 Yuxi Fu and Zhenrong Yang. Tau laws for pi calculus. *Theoretical Computer Science*, 308(1-3):55–130, 2003. doi:10.1016/S0304-3975(03)00202-0.

12 Matthew Hennessy and Huimin Lin. Symbolic bisimulations. *Theor. Comput. Sci.*, 138(2):353–389, 1995. doi:10.1016/0304-3975(94)00172-F.

13 Ross Horne, Ki Yung Ahn, Shang-Wei Lin, and Alwen Tiu. Quasi-open bisimilarity with mismatch is intuitionistic. In Anuj Dawar and Erich Grädel, editors, *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 26–35. ACM, 2018. doi:10.1145/3209108.3209125.

14 Ross Horne and Sjouke Mauw. Discovering epassport vulnerabilities using bisimilarity. *Log. Methods Comput. Sci.*, 17(2):24, 2021. doi:10.23638/LMCS-17(2:24)2021.

15 Ross Horne, Sjouke Mauw, and Semen Yurkov. When privacy fails, a formula describes an attack: A complete and compositional verification method for the applied π-calculus. *Theor. Comput. Sci.*, 959:113842, 2023. doi:10.1016/J.TCS.2023.113842.

16 Jia Liu and Huimin Lin. A complete symbolic bisimulation for full applied pi calculus. *Theor. Comput. Sci.*, 458:76–112, 2012. doi:10.1016/J.TCS.2012.07.034.

17 Tiange Liu, Alwen Tiu, and Jim de Groot. Modal logics for mobile processes revisited. In *34th International Conference on Concurrency Theory (CONCUR 2023)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2023.

18 Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i. *Information and computation*, 100(1):1–40, 1992. doi:10.1016/0890-5401(92)90008-4.

19 Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, II. *Information & Computation*, 100:41–77, 1992. doi:10.1016/0890-5401(92)90009-5.

20 Joachim Parrow and Davide Sangiorgi. Algebraic theories for name-passing calculi. *Information and Computation*, 120(2):174–197, 1995. doi:10.1006/INCO.1995.1108.

21 Joachim Parrow and Björn Victor. The fusion calculus: Expressiveness and symmetry in mobile processes. In *Proceedings. Thirteenth Annual IEEE Symposium on Logic in Computer Science (Cat. No. 98CB36226)*, pages 176–185. IEEE, 1998. doi:10.1109/LICS.1998.705654.

22 Davide Sangiorgi. A theory of bisimulation for the pi-calculus. In Eike Best, editor, *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings*, volume 715 of *Lecture Notes in Computer Science*, pages 127–142. Springer, 1993. doi:10.1007/3-540-57208-2_10.

23 Davide Sangiorgi and David Walker. On barbed equivalences in pi-calculus. In Kim Guldstrand Larsen and Mogens Nielsen, editors, *CONCUR 2001 - Concurrency Theory, 12th International Conference, Aalborg, Denmark, August 20-25, 2001, Proceedings*, volume 2154 of *Lecture Notes in Computer Science*, pages 292–304. Springer, 2001. doi:10.1007/3-540-44685-0_20.

24 Davide Sangiorgi and David Walker. *The pi-calculus: a Theory of Mobile Processes*. Cambridge university press, 2003.

25 Alwen Tiu. A trace based bisimulation for the spi calculus: An extended abstract. In Zhong Shao, editor, *Programming Languages and Systems, 5th Asian Symposium, APLAS 2007, Singapore, November 29-December 1, 2007, Proceedings*, volume 4807 of *Lecture Notes in Computer Science*, pages 367–382. Springer, 2007. doi:10.1007/978-3-540-76637-7_25.

26 Alwen Tiu and Jeremy E. Dawson. Automating open bisimulation checking for the spi calculus. In *Proceedings of the 23rd IEEE Computer Security Foundations Symposium, CSF 2010, Edinburgh, United Kingdom, July 17-19, 2010*, pages 307–321. IEEE Computer Society, 2010. doi:10.1109/CSF.2010.28.

27 Alwen Tiu and Dale Miller. Proof search specifications of bisimulation and modal logics for the pi-calculus. *ACM Trans. Comput. Log.*, 11(2):13:1–13:35, 2010. doi:10.1145/1656242.1656248.

28 Christian Urban. Nominal techniques in isabelle/hol. *J. Autom. Reason.*, 40(4):327–356, 2008. doi:10.1007/S10817-008-9097-2.

8 Appendix

8.1 Proof for Lemma 14

Proof. We illustrate the proof for transitivity here (the other cases are straightforward). By definition, $\sim$ since $P \sim^D Q$, there exists an open bisimulation relation R_1 such that $(P, Q, D) \in R_1$. Similarly, there exists a R_2 such that $(Q, V, D) \in R_2$.

Let $R_3 = \{(P, V, D) \mid (P, Q, D) \in R_1, (Q, V, D) \in R_2\}$. Obviously, $(P, V, D) \in R_3$, so it suffices to show that R_3 is an open bisimulation.

- Suppose $(P, V, D) \in R_3$, then $(P, Q, D) \in R_1$ and $(Q, V, D) \in R_2$. For all σ respecting D, we have $(P\sigma, Q\sigma, D\sigma) \in R_1$, and $(Q\sigma, V\sigma, D\sigma) \in R_2$. Then by definition of R_3, $(P\sigma, V\sigma, D\sigma) \in R_3$.
- Let D' be a distinction that $D' \supseteq D$. By definition of open bisimilarity, $(P, Q, D') \in R_1$ and $(Q, V, D') \in R_2$. Hence by definition of R_3, $(P, V, D') \in R_3$.
- If $D : P \xrightarrow{\alpha} P'$ and α is not a bound output, then $\exists Q'$ such that $D : Q \xrightarrow{\alpha} Q'$ and $(P', Q', D) \in R_1$. Then $\exists V'$ such that $D : V \xrightarrow{\alpha} V'$ and $(Q', V', D) \in R_2$. By definition of R_3, $(P', V', D) \in R_3$.
- If $D : P \xrightarrow{\bar{x}(z)} P'$ and z is fresh, then $\exists Q'$ such that $D : Q \xrightarrow{\bar{x}(z)} Q'$ and $(P', Q', D') \in R_1$ where $D' = D \cup (\{z\} \otimes \mathrm{fn}(P, Q, V))$. Then $\exists V'$ such that $D : V \xrightarrow{\bar{x}(z)} V'$ and $(Q', V', D') \in R_2$ where $D' = D \cup (\{z\} \otimes \mathrm{fn}(P, Q, V))$. By definition of R_3, $(P', V', D') \in R_3$. ◀

8.2 Proof for Lemma 21

Proof. We prove each of the conditions below (each listed as a claim followed by a proof).

▷ Claim 5. $(C[P]\sigma, C[Q]\sigma, C[D]\sigma) \in F(S)$ for each σ that respects $C[D]$.

Proof. Because $R \rightsquigarrow S$, $(P\theta, Q\theta, D\theta) \in S$ for all θ respecting D and $(C[P\theta], C[Q\theta], C[D\theta]) \in F(S)$ by definition of F.

We prove the statement by induction on C. When $C = [\cdot]$, $(C[P], C[Q], C[D]) = (P, Q, D)$. Then $C[P]\sigma = P\sigma, C[Q]\sigma = Q\sigma, C[D]\sigma = D\sigma$. Since $R \rightsquigarrow S$ and $(P, Q, D) \in R$, we have $(P\sigma, Q\sigma, D\sigma) \in S$. Thus $(C[P]\sigma, C[Q]\sigma, C[D]\sigma) \in S \subseteq F(S)$.

When $C = \nu z.C'[\cdot], C[D] = C'[D] \setminus z$ by Definition 15. Since substitutions must be capture-avoiding, without loss of generality, we can assume that z is not in the range of σ. Let $\sigma' = \sigma[z/z]$, i.e., σ' is obtained from σ by removing the mapping for z. We have that σ'

respects $C'[D] \setminus z$, and since z is a bound name $C[\cdot]$, we also have $C[P]\sigma = (\nu z.C'[P])\sigma = \nu z.(C'[P]\sigma')$ and $C[Q]\sigma = (\nu z.C'[Q])\sigma = \nu z.(C'[P]\sigma')$ and $(\nu z.C'[D])\sigma = (C'[D] \setminus z)\sigma = C'[D]\sigma'$. By induction hypothesis, $(C'[P]\sigma', C'[Q]\sigma', C'[D]\sigma') \in F(S)$. By definition of F, $(C[P]\sigma, C[Q]\sigma, C[D]\sigma) \in F(S)$.

When $C = M \mid C'[\cdot]$, by Definition 15, $C[D] = C'[D]$. By induction hypothesis, $(C'[P]\sigma, C'[Q]\sigma, C'[D]\sigma) \in F(S)$ for all σ respects $C'[D]$. By definition of F, $(M \mid C'[P]\sigma, M \mid C'[Q]\sigma, M \mid C'[D]\sigma) \in F(S)$. Because σ also respects $C[D]$, $((M \mid C'[P])\sigma, (M \mid C'[Q])\sigma, (M \mid C'[D])\sigma) \in F(S)$.

Other cases can be derived similarly. ◁

▷ Claim 6. $(C[P], C[Q], D_S) \in F(S)$ for all $D_S \supseteq C[D]$.

Proof. There are two cases to consider. In the first case, we have that the context $C[\cdot]$ does not contain any ν-binders binding a name in D_S, so we have $C[D_S] = D_S$. Let $D_1 = D \cup D_S$. Then $D_1 \supseteq D_S$ and $C[D_1] = D_S$. From the assumption that $R \rightsquigarrow S$, we have $(P, Q, D_1) \in S$, therefore $(C[P], C[Q], C[D_1]) = (C[P], C[Q], D_S) \in F(S)$ by the definition of F. For the second case, suppose that the context $C[.]$ contains a ν binder that binds a name in D_S. We consider the case where $C[.]$ binds exactly one name z in D_S (but this argument can be generalised to any number of binders). Let D_S^w be obtained from D_S by replacing z with a fresh name w. Since z is a binder in $C[.]$ it follows that $C[D]$ contains no occurrences of z. So we also have $D_S^w \supseteq C[D]$. Then by the construction in the first case, we have $(C[P], C[Q], D_S^w \in F(S)$. Now by Claim 1, we can apply the substitution $\{z/w\}$, which respects D_S^w, to obtain $(C[P], C[Q], D_S) \in F(S)$ as required. ◁

▷ Claim 7. If $C[D] : C[P] \xrightarrow{\alpha} P'$ and α is not a bound output, then $C[D] : C[Q] \xrightarrow{\alpha} Q'$ with $(P', Q', C[D]) \in F(S)$.

Proof. We prove by induction on C.

Case 1. Suppose $C = [\cdot]$, so $D : P \xrightarrow{\alpha} P'$. Since $(P, Q, D) \in R$ and $R \rightsquigarrow S$, we have $D : Q \xrightarrow{\alpha} Q'$ with $(P', Q', D) \in S$. Then $(P', Q', D) \in F(S)$ since $S \subseteq F(S)$.

Case 2. Suppose $C = \alpha.C'$ (α is not a bound output), then $P' = C'[P]$ and $C[D] = C'[D]$. $C[D] : C[Q] \xrightarrow{\alpha} C'[Q]$ by the ACT rule. Let $Q' = C'[Q]$. Because $(P, Q, D) \in R$, $(C'[P], C'[Q], C'[D]) \in F(R)$ by definition of F. Since $F(R) \subseteq F(S)$, we have $(C'[P], C'[Q], C'[D]) \in F(S)$, i.e., $(P', Q', C[D]) \in F(S)$.

Case 3. Suppose $C = \nu z.C'$, then $C[D] = C'[D]\setminus z$. By the RES rule, $C'[D]\setminus z : \nu z.C'[P] \xrightarrow{\alpha} \nu z.P'$ if $C'[D] : C'[P] \xrightarrow{\alpha} P'$. By induction hypothesis, we get $C'[D] : C'[Q] \xrightarrow{\alpha} Q'$ and $(P', Q', C'[D]) \in F(S)$. Then we have $C'[D] \setminus z : \nu z.C'[Q] \xrightarrow{\alpha} \nu z.Q'$ by the RES rule. Also by definition of F, we get $(\nu z.P', \nu z.Q', C[D]) \in F(S)$ from $(P', Q', C'[D]) \in F(S)$.

Case 4. Suppose $C = C' \mid A, A$ is any process. In this case $C[D] = C'[D]$. If $C[D] : C'[P] \mid A \xrightarrow{\alpha} P' \mid A$, then $C[D](= C'[D]) : C'[P] \xrightarrow{\alpha} P'$ according to PAR rule. By induction hypothesis, $C'[D] : C'[Q] \xrightarrow{\alpha} Q'$ and $(P', Q', C'[D]) \in F(S)$. Then we have $C[D] : C'[Q] \mid A \xrightarrow{\alpha} Q' \mid A$ by the PAR rule. From $(P', Q', C'[D]) \in F(S)$ and definition of F, we have $(P' \mid A, Q' \mid A, C[D]) \in F(S)$ as required.

Consider the symmetric case, when A is the process that is acting $C[D] : A \xrightarrow{\alpha} A'$, then $C[D] : C'[P] \mid A \xrightarrow{\alpha} C'[P] \mid A'$ and $C[D] : C'[Q] \mid A \xrightarrow{\alpha} C'[Q] \mid A'$ according to PAR rule. By definition of F, $(C'[P] \mid A', C'[Q] \mid A', C'[D] \mid A') \in F(R)$ where $C'[D] \mid A' = C[D]$. Then $(C'[P] \mid A', C'[Q] \mid A', C[D]) \in F(S)$ because $F(R) \subseteq F(S)$.

If we have $C[D] : C'[P] \xrightarrow{\bar{x}(z)} P'$; $C[D] : A \xrightarrow{x(z)} A'$, then $C[D] : C'[P] \mid A \xrightarrow{\tau} \nu z.(P' \mid A')$ according to CLOSE rule. From $C[D](= C'[D]) : C'[P] \xrightarrow{\bar{x}(z)} P'$, we have $C'[D] : C'[Q] \xrightarrow{\bar{x}(z)} Q'$ and $(P', Q', C[D]') \in F(S)$ where $C[D]' = C'[D] \cup (\{z\} \otimes \mathrm{fn}(C'[P], C'[Q]))$ by induction hypothesis. Then $C[D] : C'[Q] \mid A \xrightarrow{\tau} \nu z.(Q' \mid A')$ by the CLOSE rule. From $(P', Q', C[D]') \in F(S)$ and the definition of F, we have $(P' \mid A', Q' \mid A', C[D]') \in F(S)$ where $C[D]' = C'[D] \cup (\{z\} \otimes \mathrm{fn}(C'[P], C'[Q])) = C[D] \cup (\{z\} \otimes \mathrm{fn}(C'[P], C'[Q]))$. Then by definition of F, $(\nu z.(P' \mid A'), \nu z.(Q' \mid A'), C[Q]) \in F(S)$.

When $C[D] : C'[P] \xrightarrow{x(z)} P'$ and $C[D] : A \xrightarrow{\bar{x}y} A'$, we have $C[D] : C'[P] \mid A \xrightarrow{\tau} P'\{y/z\} \mid A'$ according to the COM rule. By induction hypothesis, $C[D](= C'[D]) : C'[Q] \xrightarrow{x(z)} Q'$ so that $C[D] : C'[Q] \mid A \xrightarrow{\tau} Q'\{y/z\} \mid A'$, and $(P', Q', C[D]) \in F(S)$. Because z is a placeholder that $z \notin C[D]$, $C[D]\{y/z\} = C[D]$. Then $(P'\{y/z\}, Q'\{y/z\}, C[D]) \in F(S)$, and $(P'\{y/z\} \mid A', Q'\{y/z\} \mid A', C[D]) \in F(S)$ by definition of F.

The symmetric case is when $C[D] : C'[P] \xrightarrow{\bar{x}y} P'$ and $C[D] : A \xrightarrow{x(z)} A'$, we have $C[D] : C'[P] \mid A \xrightarrow{\tau} P' \mid A'\{y/z\}$. By induction hypothesis, $C[D] : C'[Q] \xrightarrow{\bar{x}y} Q'$ so that $C[D] : C'[Q] \mid A \xrightarrow{\tau} Q' \mid A'\{y/z\}$ and $(P', Q', C[D]) \in F(S)$. Then by definition of F, $(P' \mid A'\{y/z\}, Q' \mid A'\{y/z\}, C[D]) \in F(S)$.

Case 5. Suppose $C = [x \neq y]C'$, then $C[D] = C'[D]$. According to the MISMATCH rule, $C[D] : [x \neq y]C'[P] \xrightarrow{\pi} P'$ when $C[D] : C'[P] \xrightarrow{\pi} P'$ and $(x, y) \in D$. By induction hypothesis, we have $C[Q] : C'[Q] \xrightarrow{\pi} Q'$ and $(P', Q', C[D]) \in F(S)$. And by the MISMATCH rule, $C[D] : [x \neq y]C'[Q] \xrightarrow{\pi} Q'$.

Other cases can be proved similarly using the same technique. ◁

▷ Claim 8. If $C[D] : C[P] \xrightarrow{\bar{x}(z)} P'$ and z is fresh, then $C[D] : C[Q] \xrightarrow{\alpha} Q'$ with $(P', Q', C[D]') \in F(S)$ where $C[D]' = C[D] \cup (\{z\} \otimes \mathrm{fn}(C[P], C[Q]))$.

Consider the bound output case in **Case 2.**, when $C = \bar{x}(z).C'$, $P' = C'[P]$. In this case $C[D] = C'[D] \setminus z$ because $\bar{x}(z)$ is just the abbreviation of $\nu z.\bar{x}z$. By the ACT rule, $C[D] : C[Q] \xrightarrow{\bar{x}(z)} C'[Q]$. Let $Q' = C'[Q]$. By definition of F, $(C'[P], C'[Q], C'[D]) \in F(R)$, i.e., $(P', Q', C[D] \cup (\{z\} \otimes \mathrm{fn}(C[P], C[Q]))) \in F(R) \subseteq F(S)$.

Consider the bound output in **Case 3.** Consider the RES rule. When $C = \nu y.C'$ and $C[D] = C'[D] \setminus y$. We have $C[D] : \nu y.C'[P] \xrightarrow{\bar{x}(z)} \nu y.P'$ if $C[D]_y : C'[P] \xrightarrow{\bar{x}(z)} P'$, where $C[D]_y = C[D] \cup (\{y\} \otimes \mathrm{fn}(C[P]))$. By induction hypothesis, $C[D]_y : C'[Q] \xrightarrow{\bar{x}(z)} Q'$ and $(P', Q', C[D]_{yz}) \in F(S)$, where $C[D]_{yz} = C[D]_y \cup (\{z\} \otimes \mathrm{fn}(C'[P], C'[Q])) = C[D]_y \cup (\{z\} \otimes \mathrm{fn}(C[P], C[Q]))$. By definition of F, we have $(\nu y.P', \nu y.Q', C[D]') \in F(S)$ where $C[D]' = C[D] \cup (\{z\} \otimes \mathrm{fn}(C[P], C[Q]))$ as required. Also by the RES rule we have $C[D] : \nu y.C'[Q] \xrightarrow{\bar{x}(z)} \nu y.Q'$.

Now we look at the OPEN rule. $C[D] : \nu z.C'[P] \xrightarrow{\bar{x}(z)} P'$ when $C[D]' : C'[P] \xrightarrow{\bar{x}z} P'$ with $z \notin \{x\} \cup D$ and $C[D]' = C[D] \cup (\{z\} \otimes \mathrm{fn}(\nu z.C'[P]))$. By induction hypothesis, $C[D]' : C'[Q] \xrightarrow{\bar{x}z} Q'$ and $(P', Q', C[D]') \in F(S)$. Also we have $C[D] : \nu z.C'[Q] \xrightarrow{\bar{x}(z)} Q'$ by the OPEN rule. ◀

Compositional Reasoning for Parametric Probabilistic Automata

Hannah Mertens[1]
RWTH Aachen University, Germany

Tim Quatmann
RWTH Aachen University, Germany

Joost-Pieter Katoen
RWTH Aachen University, Germany

Abstract

We establish an assume-guarantee (AG) framework for compositional reasoning about multi-objective queries in parametric probabilistic automata (pPA) – an extension to probabilistic automata (PA), where transition probabilities are functions over a finite set of parameters. We lift an existing framework for PA to the pPA setting, incorporating asymmetric, circular, and interleaving proof rules. Our approach enables the verification of a broad spectrum of multi-objective queries for pPA, encompassing probabilistic properties and (parametric) expected total rewards. Additionally, we introduce a rule for reasoning about monotonicity in composed pPAs.

2012 ACM Subject Classification Theory of computation → Logic

Keywords and phrases Verification, Probabilistic systems, Assume-guarantee reasoning, Parametric Probabilistic Automata, Parameter synthesis

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2025.31

Related Version *Full Version*: `https://arxiv.org/abs/2506.08525` [38]

Funding *Tim Quatmann*: This research was funded by a KI-Starter grant from the Ministerium für Kultur und Wissenschaft NRW.

1 Introduction

Probabilistic Model Checking [21, 31] studies the automated verification of Markov models for systems with random behavior. Applications include network and security protocols, biochemical processes, and planning under uncertainty [40, 33, 20]. Common properties such as reachability probabilities in Markov decision processes (MDPs) can often be verified efficiently in PTIME [3].

When the probabilities with which the system evolves are not known exactly, verification results must be robust towards slight perturbations. *Parametric Markov models* [14, 28] allow representing uncertain model quantities – for example the bias of a coin-flip or the probability of a sensor misreading – using parameters. *Feasibility* is a fundamental verification problem for parametric systems and asks whether there is an instantiation of the parameters under which a given specification holds. Deciding feasibility for reachability probability specifications in parametric MDPs is ETR-complete, i.e., at least NP-hard and within PSPACE [29]. The dual *verification* problem that asks if the specification holds under all instantiations is co-ETR complete. Checking parametric Markov models is therefore significantly more complex compared to Markov models without parameters.

[1] Corresponding author

© Hannah Mertens, Tim Quatmann, and Joost-Pieter Katoen;
licensed under Creative Commons License CC-BY 4.0

The number of system states grows exponentially with the number of system components. The resulting *state-space explosion* is an omnipresent challenge when model checking complex systems, often rendering analysis computationally infeasible. Compositional verification techniques such as *assume-guarantee* (AG) reasoning [26, 44] address this problem by decomposing the verification task into smaller sub-tasks that consider individual components in isolation. This modular verification approach has been successfully applied in various domains, including service-based workflow verification [6], large-scale IT systems [7], and autonomous systems incorporating deep neural networks [42, 43]. Recent advancements include circular AG reasoning [16] and verification-repair techniques [22]. Extensions to probabilistic systems have further expanded the scope of AG reasoning, as demonstrated in works such as [35], where AG reasoning was applied to Segala's probabilistic automata (PA) [47] – a compositional extension of Markov decision processes. Automated approaches to AG reasoning for probabilistic systems have also been explored [19, 36], enabling more scalable verification. Other works have considered parametric, but non-probabilistic timed automata [2] as well as parameterized programs [4, 48, 39] – where the concurrent system is parameterized by the number of processes or threads in a configured instance.

This work introduces a framework for compositional reasoning about parametric probabilistic automata (pPA). The case studies presented by Kwiatkowska et al. [35] demonstrate the practical applicability of AG reasoning within a non-parametric setting. These findings provide strong motivation for extending this approach to the parametric domain. In this work, we develop the theoretical foundations of an AG reasoning framework for pPAs, leveraging results from (non-parametric) PAs [35]. Due to the aforementioned ETR-hardness, compositional reasoning has a large potential and can be crucial to verify complex parameterized systems that are too large to handle monolithically.

▶ **Example 1.** Consider a communication system, where a *sender* $\mathcal{S}$ broadcasts messages to a *receiver* $\mathcal{R}$ through a *broadcast channel* $\mathcal{B}$. The system is modeled by the parallel composition $\mathcal{S} \parallel \mathcal{B} \parallel \mathcal{R}$. The components are faulty: $\mathcal{S}$ might face a collision, broadcasting in $\mathcal{B}$ might fail due to message loss, and $\mathcal{R}$ might miss broadcasts. The reliability of $\mathcal{S}$, $\mathcal{B}$, and $\mathcal{R}$ is influenced by parameters and the precise values of these parameters vary depending on network conditions, interference, or other factors. Our goal is to verify that under all parameter instantiations in a given parameter space R, the message is successfully *received* with at least probability 0.7, formally denoted by

$$(\mathcal{S} \parallel \mathcal{B} \parallel \mathcal{R}), R \models \mathbb{P}_{\geq 0.7}(\mathit{received}).$$

AG reasoning allows to verify the specification without explicitly considering the (potentially large) composition $\mathcal{S} \parallel \mathcal{B} \parallel \mathcal{R}$. To this end, assume that we have established the following statements:

- $\mathcal{S}, R \models \mathbb{P}_{<0.1}(\mathit{collision})$ – the probability that $\mathcal{S}$ faces a collision is below 0.1
- $\mathcal{B}, R \models \mathbb{P}_{<0.1}(\mathit{collision}) \rightarrow \mathbb{P}_{\geq 0.8}(\mathit{broadcast})$ – if $\mathcal{B}$ observes a collision with probability below 0.1, the message is broadcast with probability at least 0.8
- $\mathcal{R}, R \models \mathbb{P}_{\geq 0.8}(\mathit{broadcast}) \rightarrow \mathbb{P}_{\geq 0.7}(\mathit{received})$ – If the message is broadcast with probability at least 0.8, then $\mathcal{R}$ receives the message with probability at least 0.7

We reason about the composed system using the AG rule stated in Theorem 33 in Section 5:

$$\frac{\mathcal{S}, R \models \mathbb{P}_{<0.1}(\mathit{collision}) \quad \mathcal{B}, R \models \mathbb{P}_{<0.1}(\mathit{collision}) \rightarrow \mathbb{P}_{\geq 0.8}(\mathit{broadcast})}{(\mathcal{S} \parallel \mathcal{B}), R \models \mathbb{P}_{\geq 0.8}(\mathit{broadcast})} \qquad \frac{(\mathcal{S} \parallel \mathcal{B}), R \models \mathbb{P}_{\geq 0.8}(\mathit{broadcast}) \quad \mathcal{R}, R \models \mathbb{P}_{\geq 0.8}(\mathit{broadcast}) \rightarrow \mathbb{P}_{\geq 0.7}(\mathit{received})}{(\mathcal{S} \parallel \mathcal{B} \parallel \mathcal{R}), R \models \mathbb{P}_{\geq 0.7}(\mathit{received})}$$

Contributions. To the best of our knowledge, *we provide the first framework for compositional reasoning of parametric Markov models.* Our main contributions are as follows.

- We introduce and formalize pPAs, i.e., compositional probabilistic automata with parametric transitions.
- We provide a conservative extension of *strategy projections* [46, 35] to pPAs, including a more natural definition based on conditional probabilities. Strategy projections are essential for correctness of compositional reasoning as they allow to link measures of a composed model to measures of its constituting components.
- We present rules for assume-guarantee reasoning, generalizing an established framework by Kwiatkowska et al. [35] to the parametric setting – which requires some technically intricate proofs. The framework applies to ω-regular and expected total reward properties as well as multi-objective combinations thereof.
- We provide a new rule for compositional reasoning about monotonicity in pPAs. Knowing that a measure of interest – either the probability to satisfy an ω-regular specification or an expected total reward – is monotone in one or more parameters can significantly speed up verification [28, 50]. Our rule allows to derive monotonicity w.r.t. a composed pPA by only determining monotonicity for its components.

We introduce pPAs in Section 2 and discuss strategy projections in Section 3. Section 4 outlines properties of interest and Section 5 presents our AG rules. We outline results for monotonicity in Section 6 and related work in Section 7. Section 8 concludes the paper. Proofs omitted in the main part of the paper are given in the extended version [38].

2 Preliminaries

For sets X and Y, let $f\colon X \hookrightarrow Y$ denote a *partial function* from X to Y with domain $dom(f) \subseteq X$. The projection of f to a set Z is written as $f{\restriction_Z}\colon (X \cap Z) \to Y$. Iverson brackets $[\![\, \varphi \,]\!] \in \{0,1\}$ map a Boolean condition φ to 1 if φ holds and 0 otherwise.

$\mathbb{Q}[V]$ denotes the set of (multivariate) *polynomials* over a finite set of real-valued *parameters* $V = \{p_1, \ldots, p_n\}$. A (parameter) *valuation* for V is a function $v\colon V \to \mathbb{R}$. Evaluating a polynomial $f \in \mathbb{Q}[V]$ at v yields $f[v] \in \mathbb{R}$. A *region* R for V is a set of valuations. For $p \in V$, we define the valuation e_p with $e_p(q) = [\![\, p{=}q \,]\!]$ for $q \in V$.

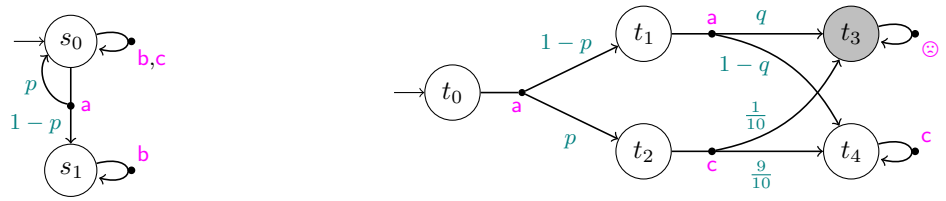
A *parametric distribution*[2] for V over a finite set S is a function $\mu\colon S \to (\mathbb{Q}[V] \cup \mathbb{R})$. Applying valuation v to μ yields $\mu[v]\colon S \to \mathbb{R}$ with $\mu[v](s) = \mu(s)[v]$ for all $s \in S$. We call $\mu\colon S \to [0,1]$ a *subdistribution* if $\sum_{s\in S} \mu(s) \leq 1$ and a *distribution* if $\sum_{s\in S} \mu(s) = 1$. The sets of parametric distributions, subdistributions, and distributions over S are denoted by $Dist_V(S)$, $SubDist(S)$, and $Dist(S)$, respectively. For $s \in S$, $\mathbb{1}_s \in Dist(S)$ is the *Dirac* distribution with $\mathbb{1}_s(s') = [\![\, s'{=}s \,]\!]$. For sets S_1, S_2, the *product* of $\mu_1 \in Dist_V(S_1)$ and $\mu_2 \in Dist_V(S_2)$ is the distribution $\mu_1 \times \mu_2 \in Dist_V(S_1 \times S_2)$ with $(\mu_1 \times \mu_2)(s_1, s_2) = \mu_1(s_1) \cdot \mu_2(s_2)$.

2.1 Parametric Probabilistic Automata

We combine probabilistic automata (PA) [47, 52] with parametric Markov models [28].

▶ **Definition 2.** *A* parametric probabilistic automaton (pPA) *over a finite alphabet Σ is a tuple $\mathcal{M} = (S, s^{init}, V, Act, \mathbf{P}, L)$, where S, V, and Act are finite sets of states, parameters, and actions, respectively, $s^{init} \in S$ is an initial state, $\mathbf{P}\colon (S \times Act) \hookrightarrow Dist_V(S)$ is a parametric transition function, and $L\colon dom(\mathbf{P}) \to \Sigma$ is a labeling function.*

[2] We use the term parametric *distribution* – rather than parametric *function* – to emphasize that we are typically interested in functions μ where $\mu[v]$ is a (sub)probability distribution.



(a) pPA $\mathcal{M}_1$. **(b)** pPA $\mathcal{M}_2$.

Figure 1 Example pPAs $\mathcal{M}_1$ and $\mathcal{M}_2$.

Let $\mathcal{M} = \big(S, s^{init}, V, Act, \mathbf{P}, L\big)$ be a pPA. For $s \in S$, $Act(s) = \{\alpha \in Act \mid (s,\alpha) \in dom(\mathbf{P})\}$ denotes the set of enabled actions in s and $\mathcal{M}_s$ is the pPA where the initial state is set to s. We set $\mathbf{P}(s,\alpha,s') = \mathbf{P}(s,\alpha)(s')$ if $(s,\alpha) \in dom(\mathbf{P})$ and otherwise $\mathbf{P}(s,\alpha,s') = 0$. $\mathcal{M}$ is a *(non-parametric)* PA if $\mathbf{P}(s,\alpha) \in Dist(S)$ for all $(s,\alpha) \in dom(\mathbf{P})$. In this case, $\mathbf{P}(s,\alpha,s')$ is the probability to transition to successor state s' when action α is selected at state s.

The *instantiation* of $\mathcal{M}$ at valuation v for V is the pPA $\mathcal{M}[v] = \big(S, s^{init}, \emptyset, Act, \mathbf{P}[v], L\big)$, where $dom(\mathbf{P}[v]) = dom(\mathbf{P})$ and $\mathbf{P}[v](s,\alpha) = \mathbf{P}(s,\alpha)[v]$. If $\mathcal{M}[v]$ is a non-parametric PA, we say v is *well-defined* for $\mathcal{M}$. A valuation v is *graph-preserving* for $\mathcal{M}$ if it is well-defined and for all $s, s' \in S$ and $\alpha \in Act_{\mathcal{M}}$: $\mathbf{P}(s,\alpha,s')[v] = 0$ iff $\mathbf{P}(s,\alpha,s') = 0$. A region R is well-defined (graph-preserving) if all its valuations $v \in R$ are well-defined (graph-preserving).

▶ **Example 3** (pPA). Consider the pPA $\mathcal{M}_1$ in Figure 1a and $\mathcal{M}_2$ in Figure 1b. $\mathcal{M}_1 = \big(S_1, s_1^{init}, V_1, Act_1, \mathbf{P}_1, L_1\big)$, where $S_1 = \{s_0, s_1\}$, $s_1^{init} = s_0$ and $V_1 = \{p\}$. Actions $Act_1 = \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$ and alphabet $\Sigma_1 = \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$. In this example, the alphabet coincides with actions as these uniquely define the transitions. Similarly, $\mathcal{M}_2 = \big(S_2, s_2^{init}, V_2, Act_2, \mathbf{P}_2, L_2\big)$, where $S_2 = \{t_0, \ldots, t_4\}$, $s_2^{init} = t_0$, and $V_2 = \{p, q\}$. Again, $Act_2 = \Sigma_2 = \{\mathsf{a}, \mathsf{c}, ☹\}$.

An *infinite path* of $\mathcal{M}$ is an alternating sequence $\pi = s_0, \alpha_0, s_1, \alpha_1, \ldots$ of states $s_i \in S$ and actions $\alpha_i \in Act$ such that $(s_i, \alpha_i) \in dom(\mathbf{P})$ for all $i \geq 0$. A finite path of length $n \in \mathbb{N}$ is a prefix $\widehat{\pi} = s_0, \alpha_0, \ldots, s_n$ of an infinite path, ending in a state $last(\widehat{\pi}) = s_n \in S$. $Paths_{\mathcal{M}}^{inf}$ and $Paths_{\mathcal{M}}^{fin}$ are the sets of infinite and finite paths of $\mathcal{M}$, respectively. For a (finite or infinite) path $\pi \in Paths_{\mathcal{M}}^{inf} \cup Paths_{\mathcal{M}}^{fin}$, we write $|\pi| \in \mathbb{N} \cup \{\infty\}$ for its length and $\pi[0,j]$ for its prefix of length $j \leq |\pi|$. We deliberately allow paths that take transitions with probability 0. As a consequence, a path of a pPA M is always also a path of any of its instantiations $\mathcal{M}[v]$ – even if v is not graph-preserving.

Strategies – also known as schedulers or adversaries – resolve nondeterminism by assigning (sub-)distributions over enabled actions based on the history – i.e., a finite path – observed so far. We allow for partial strategies that, intuitively, can choose none of the enabled actions to reflect the case that no further transition is executed.

▶ **Definition 4.** *A* (partial) strategy *for $\mathcal{M}$ is a function $\sigma\colon Paths_{\mathcal{M}}^{fin} \to SubDist(Act)$ such that $\sigma(\widehat{\pi})(\alpha) > 0$ implies $(last(\widehat{\pi}), \alpha) \in dom(\mathbf{P})$. A strategy $\sigma\colon Paths_{\mathcal{M}}^{fin} \to Dist(Act_{\mathcal{M}})$ is called* complete. *The set of all partial and complete strategies on $\mathcal{M}$ are denoted by $Str_{\mathcal{M}}^{\star}$, where $\star \in \{prt, cmp\}$, respectively. A* memoryless strategy *only depends on the last state of $\widehat{\pi}$. The set of memoryless strategies on $\mathcal{M}$ is denoted by $Str_{\mathcal{M}}^{mless,\star}$.*

For a strategy σ for $\mathcal{M}$, we may write $\sigma(\widehat{\pi}, \alpha)$ instead of $\sigma(\widehat{\pi})(\alpha)$. If σ is memoryless, we write $\sigma(s_n, \alpha)$ instead of $\sigma(\widehat{\pi}, \alpha)$, where $s_n = last(\widehat{\pi})$.

A well-defined instantiation v and a strategy σ for $\mathcal{M}$ yield a purely probabilistic process described by the *(sub)probability measure* $Pr^{v,\sigma}_{\mathcal{M}}$ on the measurable subsets of $Paths^{inf}_{\mathcal{M}}$, which is obtained by a standard cylinder set construction [3]:

$$\mathsf{Cyl}(\widehat{\pi}) = \{\pi \in Paths^{inf}_{\mathcal{M}} \mid \widehat{\pi} \text{ is a prefix of } \pi\}$$

is the *cylinder set* of a *finite* path $\widehat{\pi} = s_0, \alpha_0, \ldots, s_n$ of $\mathcal{M}$ and we set

$$Pr^{v,\sigma}_{\mathcal{M}}(\mathsf{Cyl}(\widehat{\pi})) \;=\; [\![\, s_0 = s^{init} \,]\!] \cdot \prod_{i=0}^{n-1} \sigma(\widehat{\pi}[0,i], \alpha_i) \cdot \mathbf{P}(s_i, \alpha_i, s_{i+1})[v].$$

This definition extends uniquely to a probability measure on *all* measurable sets of infinite paths. We further lift $Pr^{v,\sigma}_{\mathcal{M}}$ to (sets of) finite paths and write, e.g., $Pr^{v,\sigma}_{\mathcal{M}}(\widehat{\pi})$ for $\widehat{\pi} \in Paths^{fin}_{\mathcal{M}}$ or $Pr^{v,\sigma}_{\mathcal{M}}(\Pi)$ for $\Pi \subseteq Paths^{fin}_{\mathcal{M}}$ – implicitly referring to (unions of) cylinder sets. If $\mathcal{M}$ is a (non-parametric) PA, we may omit v and write $Pr^{\sigma}_{\mathcal{M}}$. Well-defined v yields $Pr^{\sigma}_{\mathcal{M}[v]} = Pr^{v,\sigma}_{\mathcal{M}}$.

▶ **Example 5.** For the pPA $\mathcal{M}_2$ from Figure 1b and a well-defined valuation v, the probability to reach the state t_3 under valuation v is $(1 - v(p)) \cdot v(q) + v(p) \cdot \frac{1}{10}$.

▶ Remark 6. Our definition of PA slightly deviates from related work [47, 35, 32, 36], which commonly define a transition *relation* $\delta \subseteq S \times \Sigma \times Dist(S)$ instead of functions $\mathbf{P}$ and L. In our setting, a pair $(s, \alpha) \in dom(\mathbf{P})$ uniquely identifies both, a label $L(s,\alpha) \in \Sigma$, and a distribution over successor states $\mathbf{P}(s,\alpha) \in Dist(S)$, which significantly simplifies formalizations related to pPAs. In particular, any strategy for $\mathcal{M}$ immediately also applies to instantiations of $\mathcal{M}$ and vice versa, i.e., we have $Str_{\mathcal{M}} = Str_{\mathcal{M}[v]}$ for any valuation v. On the other hand, our variant does not affect expressiveness of non-parametric PA as one can convert between the two formalisms.

We lift parallel composition of PA [47] to pPAs. Composed pPAs synchronize on common transition labels while behaving autonomously on non-common labels. For simplicity, we assume that composed pPAs consider a common set of parameters V.[3]

▶ **Definition 7** (Parallel Composition). *For $i = 1, 2$, let $\mathcal{M}_i = (S_i, s_i^{init}, V, Act_i, \mathbf{P}_i, L_i)$ be two pPAs over alphabets Σ_i with $Act_i \cap (\Sigma_1 \cup \Sigma_2) = \emptyset$. The* parallel composition *of $\mathcal{M}_1$ and $\mathcal{M}_2$ is given by the pPA $\mathcal{M}_1 \parallel \mathcal{M}_2 = \big(S_1 \times S_2, (s_1^{init}, s_2^{init}), V, Act_{\parallel}, \mathbf{P}_{\parallel}, L_{\parallel}\big)$ over $\Sigma_1 \cup \Sigma_2$, where*

- $Act_{\parallel} = (Act_1 \times Act_2) \cup (Act_1 \times \Sigma_1 \setminus \Sigma_2) \cup (\Sigma_2 \setminus \Sigma_1 \times Act_2)$,
- *for each $(s_1, \alpha_1) \in dom(\mathbf{P}_1)$, $(s_2, \alpha_2) \in dom(\mathbf{P}_2)$ with $L_1(s_1,\alpha_1) = L_2(s_2,\alpha_2) \in \Sigma_1 \cap \Sigma_2$:*

$$\mathbf{P}_{\parallel}((s_1,s_2),(\alpha_1,\alpha_2)) = \mathbf{P}_1(s_1,\alpha_1) \times \mathbf{P}_2(s_2,\alpha_2) \quad \text{and} \quad L_{\parallel}((s_1,s_2),(\alpha_1,\alpha_2)) = L_1(s_1,\alpha_1),$$

- *for each $(s_1, \alpha_1) \in dom(\mathbf{P}_1)$, $s_2 \in S_2$ with $L_1(s_1,\alpha_1) = \mathsf{a}_1 \in \Sigma_1 \setminus \Sigma_2$:*

$$\mathbf{P}_{\parallel}((s_1,s_2),(\alpha_1,\mathsf{a}_1)) = \mathbf{P}_1(s_1,\alpha_1) \times \mathbb{1}_{s_2} \quad \text{and} \quad L_{\parallel}((s_1,s_2),(\alpha_1,\mathsf{a}_1)) = \mathsf{a}_1 = L_1(s_1,\alpha_1),$$

- *for each $s_1 \in S_1$, $(s_2, \alpha_2) \in dom(\mathbf{P}_2)$ with $L_2(s_2,\alpha_2) = \mathsf{a}_2 \in \Sigma_2 \setminus \Sigma_1$:*

$$\mathbf{P}_{\parallel}((s_1,s_2),(\mathsf{a}_2,\alpha_2)) = \mathbb{1}_{s_1} \times \mathbf{P}_2(s_2,\alpha_2) \quad \text{and} \quad L_{\parallel}((s_1,s_2),(\mathsf{a}_2,\alpha_2)) = \mathsf{a}_2 = L_2(s_2,\alpha_2).$$

The parallel composition of parametric probabilistic automata (pPAs) is associative, meaning that $(\mathcal{M}_1 \parallel \mathcal{M}_2) \parallel \mathcal{M}_3$ and $\mathcal{M}_1 \parallel (\mathcal{M}_2 \parallel \mathcal{M}_3)$ are equivalent up to state renaming. Therefore, we denote this composition as $\mathcal{M}_1 \parallel \mathcal{M}_2 \parallel \mathcal{M}_3$.

[3] If two pPAs have different parameter sets $V_1 \neq V_2$, the assumption can be established by considering $V = V_1 \cup V_2$ instead, potentially adding (unused) parameters to the individual pPAs.

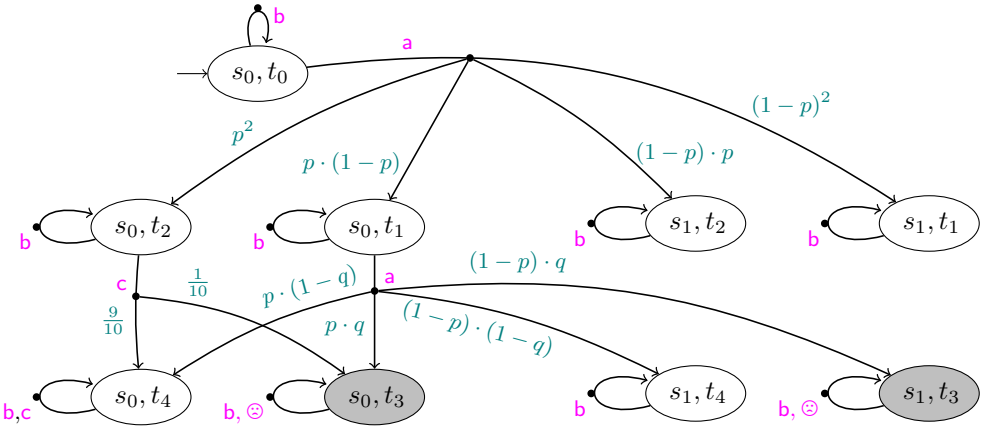


Figure 2 Parallel composition of pPAs $\mathcal{M}_1$ and $\mathcal{M}_2$ from Figure 1.

▶ **Example 8** (Parallel Composition). Figure 2 shows the composition of pPAs $\mathcal{M}_1$ and $\mathcal{M}_2$ from Figure 1. Actions with labels a or c are synchronized while b and ☹ are asynchronous.

Similar to [35, Section 3.5], we sometimes assume fairness of strategies, meaning that specific sets of labels $\Sigma_i \subseteq \Sigma$ are visited infinitely often.

▶ **Definition 9** (Fair Strategy). *Let $\mathcal{C} \subseteq 2^{\Sigma}$ and v be a well-defined valuation for $\mathcal{M}$ over Σ. A complete strategy $\sigma \in Str^{cmp}_{\mathcal{M}[v]}$ is* fair w.r.t. *$\mathcal{C}$ (denoted $fair_{\mathcal{C}}$) if*

$$Pr^{v,\sigma}_{\mathcal{M}}\Big(\big\{s_0,\alpha_0,s_1,\alpha_1,\dots \in Paths^{inf}_{\mathcal{M}} \;\big|\; \forall \Sigma_i \in \mathcal{C}\colon\ \forall j \in \mathbb{N}\colon\ \exists k \geq j\colon\ L(s_k,\alpha_k) \in \Sigma_i\big\}\Big) = 1.$$

The set of all $fair_{\mathcal{C}}$ strategies of $\mathcal{M}[v]$ is denoted $Str^{fair_{\mathcal{C}}}_{\mathcal{M}[v]}$.

Almost-sure repeated reachability in PA only depends on the graph structure, which yields:

▶ **Proposition 10.** *For any graph-preserving valuations v, v' for $\mathcal{M}$ we have $Str^{fair_{\mathcal{C}}}_{\mathcal{M}[v]} = Str^{fair_{\mathcal{C}}}_{\mathcal{M}[v']}$, i.e., a strategy is $fair_{\mathcal{C}}$ for $\mathcal{M}[v]$ iff it is $fair_{\mathcal{C}}$ for $\mathcal{M}[v']$.*

3 Strategy Projections

In this section, we define the projection of a strategy of a composite pPA $\mathcal{M} = \mathcal{M}_1 \parallel \mathcal{M}_2$ onto a single component $\mathcal{M}_i$ for $i = 1, 2$. Projections for PA are defined in [35, Definition 6] and originate from [46, page 65, Definition of Projection]. They are intuitively used to relate probability measures for $\mathcal{M}$ and $\mathcal{M}_i$.

The projection of a finite path $\pi \in Paths^{fin}_{\mathcal{M}}$ onto component $\mathcal{M}_i$ is the finite path $\pi{\restriction_i} \in Paths^{fin}_{\mathcal{M}_i}$ obtained by restricting π to the steps performed by $\mathcal{M}_i$. Formally, $(s^{init}_1, s^{init}_2){\restriction_i} = s^{init}_i$ and for $\pi = \pi', (\alpha_1, \alpha_2), (s_1, s_2)$:

$$\pi{\restriction_i} = \begin{cases} \pi'{\restriction_i}, \alpha_i, s_i & \text{if } \alpha_i \in Act_i \\ \pi'{\restriction_i} & \text{otherwise.} \end{cases}$$

Path projections are neither injective nor surjective, i.e., we might have $\pi{\restriction_i} = \pi'{\restriction_i}$ for two distinct paths $\pi \neq \pi'$ of $\mathcal{M}$, and for some $\pi_i \in Paths^{fin}_{\mathcal{M}_i}$ there might not be any $\pi \in Paths^{fin}_{\mathcal{M}}$ with $\pi_i = \pi{\restriction_i}$. We define the set of paths of $\mathcal{M}_1 \parallel \mathcal{M}_2$ that are projected to $\pi_i \in Paths^{fin}_{\mathcal{M}_i}$ as

$$\pi_i \otimes \mathcal{M}_{3-i} = \big\{\pi \in Paths^{fin}_{\mathcal{M}} \mid \pi_i = \pi{\restriction_i}\big\}.$$

We first focus on strategy projections for non-parametric PA. Then, we lift our notions to the parametric setting.

3.1 Projections for non-parametric PAs

We fix the parallel composition $\mathcal{N} = \mathcal{N}_1 \parallel \mathcal{N}_2 = \left(S_\parallel, s_\parallel^{init}, Act_\parallel, \mathbf{P}_\parallel, L_\parallel\right)$ of (non-parametric) PAs $\mathcal{N}_1$ and $\mathcal{N}_2$ with $\mathcal{N}_i = \left(S_i, s_i^{init}, Act_i, \mathbf{P}_i, L_i\right)$ and alphabets Σ_i for $i = 1, 2$.

▶ **Definition 11** (Strategy Projection for PA). *The* projection *of a strategy $\sigma \in Str_{\mathcal{N}}^{prt}$ to $\mathcal{N}_i$ is the strategy $\sigma{\restriction_i^{\mathcal{N}}} \in Str_{\mathcal{N}_i}^{prt}$, where for $\pi_i \in Paths_{\mathcal{M}_i}^{fin}$ and $\alpha_i \in Act_i$:*

$$\sigma{\restriction_i^{\mathcal{N}}}(\pi_i, \alpha_i) = \begin{cases} \dfrac{\sum_{s_i \in S_i} Pr_{\mathcal{N}}^{\sigma}((\pi_i, \alpha_i, s_i) \otimes \mathcal{N}_{3-i})}{Pr_{\mathcal{N}}^{\sigma}(\pi_i \otimes \mathcal{N}_{3-i})} & \text{if } Pr_{\mathcal{N}}^{\sigma}(\pi_i \otimes \mathcal{N}_{3-i}) > 0 \\ 0 & \text{otherwise.} \end{cases}$$

If $\mathcal{N}$ is clear, we simply write $\sigma{\restriction_i}$ instead of $\sigma{\restriction_i^{\mathcal{N}}}$. Lemmas 12 and 13 below yield that Definition 11 is equivalent to the projection defined in [35, Def. 6]. We argue that our variant is more intuitive since the numerator and denominator of the fraction consider the same probability measure $Pr_{\mathcal{N}}^{\sigma}$. Intuitively, $\sigma{\restriction_i}(\pi_i, \alpha_i)$ coincides with the conditional probability that – under $Pr_{\mathcal{N}}^{\sigma}$ and given that a path π with projection $\pi{\restriction_i} = \pi_i$ is observed – the next action of component $\mathcal{N}_i$ is α_i. We now provide an alternative characterization for the numerator given in Definition 11.

▶ **Lemma 12.** *For $\sigma \in Str_{\mathcal{N}}^{prt}$, $\pi_i \in Paths_{\mathcal{N}_i}^{fin}$, and $\alpha_i \in Act_i$:*

$$\sum_{s_i \in S_i} Pr_{\mathcal{N}}^{\sigma}((\pi_i, \alpha_i, s_i) \otimes \mathcal{N}_{3-i}) = \sum_{\pi \in (\pi_i \otimes \mathcal{N}_{3-i})} \sum_{(\hat{\alpha}_1, \hat{\alpha}_2) \in Act_\parallel,\, \hat{\alpha}_i = \alpha_i} Pr_{\mathcal{N}}^{\sigma}(\pi) \cdot \sigma(\pi, (\hat{\alpha}_1, \hat{\alpha}_2)).$$

The next lemma is the key observation for strategy projections as it connects the probability measure $Pr_{\mathcal{N}}^{\sigma}$ for $\mathcal{N}$ and $Pr_{\mathcal{N}_i}^{\sigma{\restriction_i}}$ for each component $\mathcal{N}_i$.

▶ **Lemma 13.** *For $\sigma \in Str_{\mathcal{N}}^{prt}$ and $\pi_i \in Paths_{\mathcal{N}_i}^{fin}$: $Pr_{\mathcal{N}_i}^{\sigma{\restriction_i}}(\pi_i) = Pr_{\mathcal{N}}^{\sigma}(\pi_i \otimes \mathcal{N}_{3-i})$.*

The following result lifts [35, Lemma 2] to our setting and states that fair (and therefore also complete) strategies have fair and complete projections.

▶ **Lemma 14.** *Let $\mathcal{C}_1 \subseteq 2^{\Sigma_1}$ and $\mathcal{C}_2 \subseteq 2^{\Sigma_2}$. If $\sigma \in Str_{\mathcal{N}}^{fair_{\mathcal{C}_1 \cup \mathcal{C}_2}}$, then $\sigma{\restriction_i} \in Str_{\mathcal{N}_i}^{fair_{\mathcal{C}_i}}$.*

The converse of Lemma 14 does not hold: The projection $\sigma{\restriction_i}$ might not be a complete strategy for $\mathcal{N}_i$, even though σ is complete for $\mathcal{N}$. Furthermore, $\sigma{\restriction_i}$ might not be memoryless, even if σ is memoryless. The following example shows both cases.

▶ **Example 15** (Strategy Projection for pPA). Consider the pPA $\mathcal{M} = \mathcal{M}_1 \parallel \mathcal{M}_2$ from Figure 2, using $\mathcal{M}_1$ and $\mathcal{M}_2$ depicted in Figure 1. We fix the valuation v with $v(p) = v(q) = 0.1$ and set $\mathcal{N} = \mathcal{M}[v]$ and $\mathcal{N}_i = \mathcal{M}_i[v]$ for $i = 1, 2$. Let $\sigma \in Str_{\mathcal{N}_1 \parallel \mathcal{N}_2}^{mless,cmp}$ that always selects the actions with label a, c, or ☹ with probability 1 when available; otherwise, it chooses b.

We compute the projection $\sigma{\restriction_2}$ to the PA $\mathcal{M}_2[v]$:

- $\sigma{\restriction_2}(t_0, \mathsf{a}) = 1$
- $\sigma{\restriction_2}((t_0, \mathsf{a}, t_2), \mathsf{c}) = \frac{(v(p))^2}{v(p)} = v(p) = 0.1$
- $\sigma{\restriction_2}((t_0, \mathsf{a}, t_2, \mathsf{c}, t_3), ☹) = \frac{(v(p))^2 \cdot \frac{1}{10}}{v(p) \cdot \frac{1}{10}} = v(p) = 0.1$

Similarly, $\sigma{\restriction_2}((t_0, \mathsf{a}, t_1), \mathsf{a}) = v(p) = 0.1$ and $\sigma{\restriction_2}((t_0, \mathsf{a}, t_1, \mathsf{a}, t_3), ☹) = v(p) = 0.1$. We observe that the projection is not a complete strategy as ☹ is the only available action in t_3 for $\mathcal{M}_2$. Moreover, the projection is not memoryless, because, for example, $\sigma{\restriction_2}((t_0, \mathsf{a}, t_2, \mathsf{c}, t_4), \mathsf{c}) = v(p) = 0.1$ is not equal to $\sigma{\restriction_2}((t_0, \mathsf{a}, t_1, \mathsf{a}, t_4), \mathsf{c}) = 1$.

3.2 Projections for Parametric PAs

We now lift strategy projections to the parametric case. We fix the parallel composition $\mathcal{M} = \mathcal{M}_1 \parallel \mathcal{M}_2$ of two pPAs $\mathcal{M}_1$ and $\mathcal{M}_2$ with a common set of parameters V. Let $i = 1, 2$ and let $v_i \colon V \to \mathbb{R}$ be a well-defined valuation for $\mathcal{M}_i$. We define strategy projections for pPAs in terms of the instantiated PAs.

▶ **Definition 16** (Strategy Projection for pPA). *The* projection *of a strategy $\sigma \in Str^{prt}_{\mathcal{M}}$ to pPA $\mathcal{M}_i$ w.r.t. v_1, v_2 is the strategy $\sigma{\restriction}_i^{v_1,v_2} \in Str^{prt}_{\mathcal{M}_i}$, defined by $\sigma{\restriction}_i^{v_1,v_2} = \sigma{\restriction}_i^{\mathcal{M}_1[v_1]\parallel\mathcal{M}_2[v_2]}$.*

If the valuations coincide, i.e., $v_1 = v_2 = v$, we write $\sigma{\restriction}_i^{v}$ instead of $\sigma{\restriction}_i^{v_1,v_2}$. Strategy projections depend on the parameter instantiations as the following example illustrates. Such strategies are also referred to as *(parameter) dependent strategies* [49, Def. 2.6].

▶ **Example 17** (Strategy Projection for pPA). Consider $\mathcal{M}_1 \parallel \mathcal{M}_2$ from Figure 2 and the strategy σ as in Example 15. Let v_1 and v_2 be the valuations used for $\mathcal{M}_1$ and $\mathcal{M}_2$, respectively, with $v_1(p) = v_1(q) = 0.1$, and $v_2(p) = v_2(q) = 0.9$. For the projection $\sigma{\restriction}_2^{v_1,v_2}$ to $\mathcal{M}_2$, the following values are obtained:

- $\sigma{\restriction}_2^{v_1,v_2}(t_0, \mathsf{a}) = 1$
- $\sigma{\restriction}_2^{v_1,v_2}((t_0, \mathsf{a}, t_2), \mathsf{c}) = \frac{v_1(p)\cdot v_2(p)}{v_2(p)} = v_1(p) = 0.1$
- $\sigma{\restriction}_2^{v_1,v_2}((t_0, \mathsf{a}, t_2, \mathsf{c}, t_3), ☹) = \frac{v_1(p)\cdot v_2(p)\cdot \frac{1}{10}}{v_2(p)\cdot \frac{1}{10}} = v_1(p) = 0.1,$
- $\sigma{\restriction}_2^{v_1,v_2}((t_0, \mathsf{a}, t_1), \mathsf{a}) = v_1(p) = 0.1$, and
- $\sigma{\restriction}_2^{v_1,v_2}((t_0, \mathsf{a}, t_1, \mathsf{a}, t_3), ☹) = v_1(p) = 0.1.$

Note that the resulting projection coincides with the one computed in Example 15, where both components were instantiated using v_1, i.e., $\sigma{\restriction}_2^{v_1,v_2} = \sigma{\restriction}_2^{v_1} = \sigma{\restriction}_2^{v_1,v_1}$.

The next lemma states that – when restricting to valuations that yield the same non-zero transitions – the strategy projection to $\mathcal{M}_i$ only depends on the parameter instantiation applied to $\mathcal{M}_{3-i}$. This observation is the key insight for the correctness of the proof rule in Theorem 41, which enables compositional reasoning about monotonicity.

▶ **Lemma 18.** *For $i = 1, 2$, and well-defined valuations $v_i, v_i' \colon V \to \mathbb{R}$ for $\mathcal{M}_i$ such that $\mathbf{P}_i(s_i, \alpha_i, s_i')[v_i] = 0$ iff $\mathbf{P}_i(s_i, \alpha_i, s_i')[v_i'] = 0$ we have: $\sigma{\restriction}_1^{v_1,v_2} = \sigma{\restriction}_1^{v_1',v_2}$ and $\sigma{\restriction}_2^{v_1,v_2} = \sigma{\restriction}_2^{v_1,v_2'}$.*

4 Verification Objectives for pPAs

We define properties of interest for pPA verification. Let Σ be a finite alphabet. $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$ denotes the set of all finite and infinite words over Σ. For a word $\rho = \mathsf{a}_0, \mathsf{a}_1, \cdots \in \Sigma^\infty$ and another alphabet $\widehat{\Sigma}$, let $\rho{\restriction}_{\widehat{\Sigma}} \in \widehat{\Sigma}^\infty$ denote the projection of ρ onto $\widehat{\Sigma}$ – obtained by dropping all $\mathsf{a}_i \in \Sigma \setminus \widehat{\Sigma}$ from ρ. The restriction $\rho{\restriction}_{\widehat{\Sigma}}$ can be finite, even if ρ is infinite.

We fix a pPA $\mathcal{M} = \big(S, s^{init}, V, Act, \mathbf{P}, L\big)$. The trace of $\pi = s_0, \alpha_0, s_1, \alpha_1, \cdots \in Paths^{inf}_{\mathcal{M}}$ is the sequence $tr(\pi) = L(s_0, \alpha_0), L(s_1, \alpha_1), \ldots$ of transition labels. The *probability of a language* $\mathcal{L} \subseteq \Sigma^\infty$ at a well-defined valuation v under strategy σ of $\mathcal{M}$ is given by

$$Pr^{v,\sigma}_{\mathcal{M}}(\mathcal{L}) = Pr^{v,\sigma}_{\mathcal{M}}\Big(\{\pi \in Paths^{inf}_{\mathcal{M}} \mid tr(\pi){\restriction}_\Sigma \in \mathcal{L}\}\Big).$$

We also consider (parametric) *expected total reward properties*. Let V be a set of parameters. A *reward function* $\mathcal{R} \colon \Sigma \to \mathbb{Q}[V] \cup \mathbb{R}_{\geq 0}$ over Σ assigns a (potentially parametric) reward to each symbol $\mathsf{a} \in \Sigma$. *Instantiation* of $\mathcal{R}$ at a valuation $v \colon V \to \mathbb{R}$ yields $\mathcal{R}[v]$ with $\mathcal{R}[v](\mathsf{a}) = \mathcal{R}(\mathsf{a})[v]$ for all $\mathsf{a} \in \Sigma$. Valuation v is *well-defined* for $\mathcal{R}$ if $\mathcal{R}[v] \colon \Sigma \to \mathbb{R}_{\geq 0}$. In this case, the accumulated reward for a word $\rho = \mathsf{a}_0, \mathsf{a}_1, \cdots \in \Sigma^\infty$ is given by $\mathcal{R}[v](\rho) = \sum_{i=0}^{|\rho|} \mathcal{R}[v](\mathsf{a}_i) \in \mathbb{R}_{\geq 0} \cup \{\infty\}$.

When applied to a pPA $\mathcal{M}$, a reward function $\mathcal{R}$ assigns the reward $\mathcal{R}(L(s,\alpha))$ to the enabled state-action-pairs $(s,\alpha) \in dom(\mathbf{P})$ with $L(s,\alpha) \in \Sigma$. For a well-defined valuation v for $\mathcal{M}$ and $\mathcal{R}$, we define the *expected total reward* under strategy σ as

$$Ex^{v,\sigma}_{\mathcal{M}}(\mathcal{R}) = \int_{\pi \in Paths^{inf}_{\mathcal{M}}} \mathcal{R}[v]\big(tr(\pi)\restriction_{\Sigma}\big)\, dPr^{v,\sigma}_{\mathcal{M}}(\pi)\,.$$

We consider probabilistic and reward-based objectives as well as their multi-objective combinations.

▶ **Definition 19** (Objectives). *For $\sim \in \{>, \geq, <, \leq\}$, $p \in [0,1]$, and $r \in \mathbb{R}_{\geq 0}$, we denote*

- *a* probabilistic objective *over $\mathcal{L} \subseteq \Sigma^\infty$ by $\mathbb{P}_{\sim p}(\mathcal{L})$ and*
- *a* reward objective *over $\mathcal{R}\colon \Sigma \to \mathbb{Q}[V] \cup \mathbb{R}_{\geq 0}$ by $\mathbb{E}_{\sim r}(\mathcal{R})$.*

Their satisfaction for a well-defined valuation v and strategy σ is defined by

$$\mathcal{M}, v, \sigma \models \mathbb{P}_{\sim p}(\mathcal{L}) \;\Leftrightarrow\; Pr^{v,\sigma}_{\mathcal{M}}(\mathcal{L}) \sim p \quad \text{and} \quad \mathcal{M}, v, \sigma \models \mathbb{E}_{\sim r}(\mathcal{R}) \;\Leftrightarrow\; Ex^{v,\sigma}_{\mathcal{M}}(\mathcal{R}) \sim r.$$

Let $\varphi \in \{\mathbb{P}_{\sim p}(\mathcal{L}), \mathbb{E}_{\sim r}(\mathcal{R})\}$ refer to a (probabilistic or reward) objective. If neither $\mathcal{M}$ nor φ consider any parameters, we may drop the valuation from the notation and just write $\mathcal{M}, \sigma \models \varphi$. We lift the satisfaction relation $\models$ to regions, i.e., sets of valuations.

▶ **Definition 20** (Region Satisfaction Relation). *Let $\star \in \{prt, cmp\} \cup \{fair_{\mathcal{C}} \mid \mathcal{C} \subseteq 2^{\Sigma_{\mathcal{M}}}\}$. For objective φ and well-defined region R for $\mathcal{M}$ – and $\mathcal{R}$ if $\varphi = \mathbb{E}_{\sim r}(\mathcal{R})$ – the* region satisfaction relation $\models^\star$ *is given by:*

$$\mathcal{M}, R \models^\star \varphi \quad \Leftrightarrow \quad \forall v \in R : \forall \sigma \in Str^\star_{\mathcal{M}} : \mathcal{M}, v, \sigma \models \varphi.$$

Satisfaction under memoryless strategies – denoted by $\models^{mless,\star}$ – is defined similarly.

▶ Remark 21. For $\star \in \{prt, cmp\}$ and any well-defined valuation v we have $Str^\star_{\mathcal{M}[v]} = Str^\star_{\mathcal{M}}$. Thus, for well-defined R, we can swap the quantifiers in Definition 20:

$$\mathcal{M}, R \models^\star \varphi \quad \Leftrightarrow \quad \forall \sigma \in Str^\star_{\mathcal{M}} : \forall v \in R : \mathcal{M}, v, \sigma \models \varphi.$$

However, this is not the case for *fair* strategies and regions that are not graph-preserving: A strategy that is not $fair_{\mathcal{C}}$ for $\mathcal{M}$ (under graph-preserving instantiations) might be $fair_{\mathcal{C}}$ for $\mathcal{M}[v]$ if v is not graph-preserving, because states that violate the fairness condition might not be reachable in $\mathcal{M}[v]$. For a *graph-preserving* region R and all $v \in R$, we have $Str^{fair_{\mathcal{C}}}_{\mathcal{M}} = Str^{fair_{\mathcal{C}}}_{\mathcal{M}[v]}$. Thus, we can swap quantifiers as above.

Our framework also handles conjunctions of multiple objectives.

▶ **Definition 22** (MO-Query). *A* multi-objective *query (mo-query) is a set $\mathsf{X} = \{\varphi_1, \ldots, \varphi_n\}$ of n probabilistic or reward objectives with $\mathcal{M}, v, \sigma \models \mathsf{X} \;\Leftrightarrow\; \mathcal{M}, v, \sigma \models \varphi_i$ for all $\varphi_i \in \mathsf{X}$.*

The conjunction of two mo-queries is a union of sets: $\mathsf{X}_1 \wedge \mathsf{X}_2 = \mathsf{X}_1 \cup \mathsf{X}_2$. We lift objective satisfaction for regions (Definition 20) to mo-queries in a straightforward way.

▶ Remark 23. In [38] it is shown that model checking under partial strategies in $\mathcal{M}$ for probabilistic properties, rewards, and multi-objective queries reduces to model checking under complete strategies in a modified pPA, denoted $\mathcal{M}_\tau$. This result extends [35, Proposition 2] to pPA while preserving memorylessness of the strategies.

We consider *safety objectives* as a special type of probabilistic objectives.

▶ **Definition 24** (Safety Objective). $\mathbb{P}_{\geq p}(\mathcal{L})$ *is a* safety objective[4] *if* $\mathcal{L}$ *can be characterized by a DFA* $\mathcal{A}^{bad}_{\mathcal{L}}$ *accepting a language of finite words (bad prefixes):*

$$\mathcal{L} = \{w \in \Sigma^\infty \mid \textit{no prefix of } w \textit{ is accepted by } \mathcal{A}^{bad}_{\mathcal{L}}\}.$$

A mo-query is called *safe*, denoted X^{safe}, if each φ_i is a probabilistic safety objective.

For PA, computing the probability for a safety objective reduces to maximal reachability properties in the PA-DFA product [35, Lemma 1]. This result can be lifted to pPA in a straightforward manner; see [38]. For reachability and safety objectives, it is equivalent to quantify over complete or partial strategies. Lemma 25 lifts [35, Proposition 1] to pPA.

▶ **Lemma 25.** *Let* $\mathcal{M}$ *be a pPA, let* R *be a well-defined region and let* $\mathbb{P}_{\geq p}(\mathcal{L})$ *be a safety objective for* $\mathcal{M}$*. It holds that:* $\mathcal{M}, R \models^{cmp} \mathbb{P}_{\geq p}(\mathcal{L}) \Leftrightarrow \mathcal{M}, R \models^{prt} \mathbb{P}_{\geq p}(\mathcal{L})$*. Same for* $\models^{mless,\star}$*.*

4.1 Preservation Under Projection

We generalize the result from [35, Lemma 3] – originally stated in [46, Lemma 7.2.6] – to the parametric setting. In particular, we show that probabilistic and reward properties in a composed pPA $\mathcal{M} = \mathcal{M}_1 \parallel \mathcal{M}_2$ under a strategy σ are preserved when projecting to either component $\mathcal{M}_i$ over Σ_i, assuming a well-defined valuation v.

▶ **Theorem 26.** *For* $i = 1, 2$*, let* $\mathcal{L}$ *be a language over* Σ_i *and* $\mathcal{R}$ *be a reward function over* Σ_i*. Then, for a well-defined valuation* v*:*

$$Pr^{v,\sigma\restriction_i^v}_{\mathcal{M}_i}(\mathcal{L}) = Pr^{v,\sigma}_{\mathcal{M}_1\parallel\mathcal{M}_2}(\mathcal{L}) \quad \textit{and} \quad Ex^{v,\sigma\restriction_i^v}_{\mathcal{M}_i}(\mathcal{R}) = Ex^{v,\sigma}_{\mathcal{M}_1\parallel\mathcal{M}_2}(\mathcal{R})$$

▶ **Example 27.** Let $\mathcal{L} = \{w \in \{\mathsf{a}, \mathsf{c}, ☹\}^\infty \mid \exists i \in \mathbb{N} : w_i = ☹\}$ be the language of words in which ☹ occurs. Reconsider the strategy σ of $\mathcal{M}_1 \parallel \mathcal{M}_2$ from Example 17 and the projection $\sigma\restriction_2^v$ to $\mathcal{M}_2$. We have $Pr^{v,\sigma}_{\mathcal{M}_1\parallel\mathcal{M}_2}(\mathcal{L}) = Pr^{v,\sigma\restriction_2^v}_{\mathcal{M}_2}(\mathcal{L}) = (v(p))^2 \cdot \frac{1}{10} + v(p) \cdot (1 - v(p)) \cdot v(q))$.

Theorem 26 assumes that the property only involves action labels from a single component $\mathcal{M}_i$. To allow objectives over arbitrary alphabets Σ, we can add a self-loop labeled a at every state, for each label $\mathsf{a} \notin \Sigma_i$.

▶ **Definition 28** (Alphabet Extension). *Let* $\mathcal{M} = \big(S, s^{init}, V, Act, \mathbf{P}, L\big)$ *be a pPA over* $\Sigma_\mathcal{M}$ *and let* Σ *be an alphabet with* $Act \cap (\Sigma \setminus \Sigma_\mathcal{M}) = \emptyset$*. The* alphabet extension *of* $\mathcal{M}$ *with respect to* Σ *is the pPA* $\mathcal{M}\langle\Sigma\rangle = (S, s^{init}, V, Act \cup (\Sigma \setminus \Sigma_\mathcal{M}), \mathbf{P}_\Sigma, L_\Sigma)$ *over alphabet* $\Sigma_\mathcal{M} \cup \Sigma$*, where*

- $\mathbf{P}_\Sigma(s, \alpha) = \mathbf{P}(s, \alpha)$ *and* $L_\Sigma(s, \alpha) = L(s, \alpha)$ *for all* $(s, \alpha) \in dom(\mathbf{P})$ *and*
- $\mathbf{P}_\Sigma(s, \mathsf{a}) = \mathbb{1}_s$ *and* $L_\Sigma(s, \mathsf{a}) = \mathsf{a}$ *for all* $s \in S$ *and* $\mathsf{a} \in \Sigma \setminus \Sigma_\mathcal{M}$*.*

▶ **Example 29** (Alphabet Extension). Figure 3 shows $\mathcal{M}_2\langle\{\mathsf{a}, \mathsf{b}\}\rangle$ for pPA $\mathcal{M}_2$ over $\Sigma_2 = \{\mathsf{a}, \mathsf{c}, ☹\}$ from Figure 1b. Transitions with label a remain unchanged as $\mathsf{a} \in \Sigma_2$, but an additional self-loop with action $\mathsf{b} \notin \Sigma_2$ is added to every state.

We now lift [35, Lemma 3] to the parametric setting, covering properties and mo-queries over an alphabet that is not necessarily shared by $\mathcal{M}_i$:

▶ **Theorem 30.** *Let* $\Sigma \subseteq \Sigma_{\mathcal{M}_1\parallel\mathcal{M}_2}$*, and* σ *be a strategy for* $\mathcal{M}_1\langle\Sigma\rangle \parallel \mathcal{M}_2\langle\Sigma\rangle$*. Let* $\mathcal{L}$ *be a language over* Σ *and* $\mathcal{R}$ *be a reward function over* Σ*. Then, for well-defined valuation* v*:*

$$Pr^{v,\sigma\restriction_i^v}_{\mathcal{M}_i\langle\Sigma\rangle}(\mathcal{L}) = Pr^{v,\sigma}_{\mathcal{M}_1\langle\Sigma\rangle\parallel\mathcal{M}_2\langle\Sigma\rangle}(\mathcal{L}) \quad \textit{and} \quad Ex^{v,\sigma\restriction_i^v}_{\mathcal{M}_i\langle\Sigma\rangle}(\mathcal{R}) = Ex^{v,\sigma}_{\mathcal{M}_1\langle\Sigma\rangle\parallel\mathcal{M}_2\langle\Sigma\rangle}(\mathcal{R})$$

[4] Note that safety objectives contain all finite prefixes of words in $\mathcal{L}$, i.e., they are prefix-closed. This is different in [35], where only infinite words are considered, leading to technical problems. See [38].

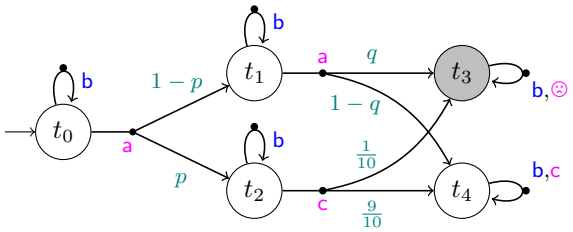


Figure 3 Alphabet extension $\mathcal{M}_2\langle\{\mathsf{a},\mathsf{b}\}\rangle$ of the pPA $\mathcal{M}_2$ from Figure 1b to the alphabet $\{\mathsf{a},\mathsf{b}\}$.

Let X be a mo-query over Σ. Then, for any well-defined valuation v:

$$\mathcal{M}_i\langle\Sigma\rangle, v, \sigma\restriction_i^v \models X \quad\Leftrightarrow\quad (\mathcal{M}_1\langle\Sigma\rangle \parallel \mathcal{M}_2\langle\Sigma\rangle), v, \sigma \models X$$

▶ Remark 31. Since alphabet extensions add self-loop transitions for new labels, and thus do not change the system's state, the pPAs $\mathcal{M}_1\langle\Sigma\rangle \parallel \mathcal{M}_2\langle\Sigma\rangle$ and $\mathcal{M}_1 \parallel \mathcal{M}_2$ satisfy the same properties and mo-queries over the alphabet $\Sigma \subseteq \Sigma_{\mathcal{M}_1\parallel\mathcal{M}_2}$.

Theorems 26 and 30 play a key role in the proof of the AG framework for reasoning about mo-queries and monotonicity, which will be established in Sections 5 and 6.

5 Assume-Guarantee Reasoning for pPA

Kwiatkowska et al. [35] introduced assume-guarantee (AG) reasoning proof rules for PA. This section extends their proof rules to the parametric setting. We first generalize the concept of AG triples to pPAs in Section 5.1. Then, we extend the asymmetric and circular proof rule in Section 5.2. Additional proof rules from [35] are presented in [38].

5.1 Assume-Guarantee Triples for pPA

We extend compositional reasoning to the parametric setting by generalizing assume-guarantee (AG) triples. Intuitively, an AG triple states that if a component satisfies an assumption, it also satisfies the guarantee under the same strategy and valuation.

▶ **Definition 32** (AG Triple). *The* assume-guarantee triple *for $\mathcal{M}$, (parametric) mo-queries A (assumption) and G (guarantee), well-defined region R, and $\star \in \{cmp, prt, fair_{\mathcal{C}}\}$ is*

$$\mathcal{M}, R \models^{\star} A{\rightarrow}G \quad\Leftrightarrow\quad \Big(\forall v \in R : \forall \sigma \in Str^{\star}_{\mathcal{M}[v]} : \quad \mathcal{M}, v, \sigma \models A \quad\rightarrow\quad \mathcal{M}, v, \sigma \models G\Big)$$

5.2 Assume-Guarantee Rules for pPA

We present AG proof rules for the compositional verification of parametric probabilistic automata (pPAs). In the remainder of this section, we fix two pPAs $\mathcal{M}_1$ and $\mathcal{M}_2$ with alphabets Σ_1, and Σ_2, respectively. Further, let R_i be a well-defined region for $\mathcal{M}_i$.

First, we establish the asymmetric proof rule for safety and mo-queries – based on [35, Theorem 1 and 2], respectively – for pPA.

▶ **Theorem 33** (Asymmetric Rule). *Let A and G be mo-queries over $\Sigma_A \subseteq \Sigma_1$ and $\Sigma_G \subseteq \Sigma_2 \cup \Sigma_A$, respectively. Let $\mathcal{C}_1 \subseteq 2^{\Sigma_1}$ and $\mathcal{C}_2 \subseteq 2^{\Sigma_2 \cup \Sigma_A}$. Then, the two proof rules holds:*

$$\frac{\mathcal{M}_1, R_1 \models^{cmp} A^{safe} \qquad \mathcal{M}_2\langle\Sigma_{A^{safe}}\rangle, R_2 \models^{prt} A^{safe}{\rightarrow}G^{safe}}{\mathcal{M}_1 \parallel \mathcal{M}_2, R_1 \cap R_2 \models^{cmp} G^{safe}} \qquad \frac{\mathcal{M}_1, R_1 \models^{fair_{\mathcal{C}_1}} A \qquad \mathcal{M}_2\langle\Sigma_A\rangle, R_2 \models^{fair_{\mathcal{C}_2}} A{\rightarrow}G}{\mathcal{M}_1 \parallel \mathcal{M}_2, R_1 \cap R_2 \models^{fair_{\mathcal{C}_1 \cup \mathcal{C}_2}} G}$$

Proof sketch. Let $v \in R_1 \cap R_2$, and σ be a strategy for the composed pPA $\mathcal{M}_1 \parallel \mathcal{M}_2$. To prove validity of the rule, we need to show that $\mathcal{M}_1 \parallel \mathcal{M}_2$ instantiated with v satisfies G^{safe}.

1. Since $v \in R_1$, the first premise implies $\mathcal{M}_1, v \models^{cmp} \mathsf{A}^{safe}$, which is equivalent to $\mathcal{M}_1, v \models^{prt} \mathsf{A}^{safe}$ by Lemma 25. This implies that $\mathcal{M}_1$ under the partial strategy $\sigma\restriction^{v}_{\mathcal{M}_1}$ also satisfies A^{safe}. Since strategies and their projections satisfy the same properties (as shown in Theorem 30), we conclude that $\mathcal{M}_1 \parallel \mathcal{M}_2$ instantiated at v under the strategy σ satisfies A^{safe}.
2. As $v \in R_2$, the second premise implies that $\mathcal{M}_2[v]$ under the strategy $\sigma\restriction^{v}_{\mathcal{M}_2}$ satisfies G^{safe}. Again, Theorem 30 implies that $(\mathcal{M}_1 \parallel \mathcal{M}_2)[v]$ under the strategy σ satisfies G^{safe}.

Thus, we conclude that $\mathcal{M}_1 \parallel \mathcal{M}_2$ instantiated at v under σ satisfies G^{safe}. The rule on the right holds by a similar reasoning, where, in addition, Lemma 14 is used to establish that projections of fair strategies remain fair. ◀

▶ **Example 34** (Asymmetric Rule)**.** We illustrate the left proof rule from Theorem 33 for the pPA $\mathcal{M}_1 \parallel \mathcal{M}_2$ in Figure 2 – composed of the pPAs $\mathcal{M}_1$ and $\mathcal{M}_2$ depicted in Figure 1 – and w.r.t. $\mathsf{G} = \mathbb{P}_{\geq 0.8}(\mathcal{L}_G)$, where $\mathcal{L}_G = \{w \in \{\mathsf{a},\mathsf{b},\mathsf{c},☹\}^\infty \mid |w|_{☹} = 0\}$. Let $\mathsf{A} = \{\mathbb{P}_{\geq 0.9}(\mathcal{L}_A)\}$, where $\mathcal{L}_A = \{w \in \{\mathsf{a},\mathsf{b}\}^\infty \mid |w|_{\mathsf{a}} \leq 1\}$. The pPA $\mathcal{M}_2\langle\{\mathsf{a},\mathsf{b}\}\rangle$ is depicted in Figure 3. For the premises of the proof rule, we obtain that the (largest) region R for which $\mathcal{M}_1, R_1 \models^{cmp} \mathbb{P}_{\geq 0.9}(\mathcal{L}_A)$ is $R_1 = \{v\colon \{p,q\} \to \mathbb{R} \mid v(p) \in [0, 0.1]\}$ and the (largest) region R_2 for which $\mathcal{M}_2\langle\{\mathsf{a},\mathsf{b}\}\rangle, R_2 \models^{prt} \mathsf{A} \to \mathsf{G}$ holds, is $R = \{v\colon \{p,q\} \to \mathbb{R} \mid (v(p) \in [0, 0.5], v(q) \in [0,1]) \vee (v(p) \in (0.5, 1), v(q) \in [0, 2 - 2 \cdot p])\}$. The intersection $R_1 \cap R_2$ – for which $\mathcal{M}_1 \parallel \mathcal{M}_2, R_1 \cap R_2 \models^{prt} \mathsf{G}$ holds by [38, Theorem 52] – contains all valuations with $v(p) \in [0, 0.1], v(q) \in [0, 1]$. The (largest) region R for which $\mathcal{M}_1 \parallel \mathcal{M}_2, R \models^{prt} \mathsf{G}$ is $R = \{v\colon p, q \to \mathbb{R} \mid (v(p) \in ([0, \frac{1}{4}] \cup \{1\}), v(q) \in [0,1]) \vee (v(p) \in (\frac{1}{4}, 1), v(q) \in [0, \frac{p+1}{5 \cdot p}])\}$. This satisfies $(R_1 \cap R_2) \subset R$.

The proof rules in Theorem 33 can be extended to systems with more than two components, as detailed in [38, Theorem 52]. Next, we lift the circular proof rule given in [35, Theorem 5] to pPAs:

▶ **Theorem 35** (Circular Rule)**.** *Let A_1,A_2 and G be (parametric) mo-queries over $\Sigma_{A_1} \subseteq \Sigma_2$, $\Sigma_{A_2} \subseteq \Sigma_1 \cup \Sigma_{A_1}$ and $\Sigma_G \subseteq \Sigma_2 \cup \Sigma_{A_2}$, respectively. Let $\mathcal{C}_i \in 2^{\Sigma_i \cup \Sigma_{A_i}}$ for $i \in \{1,2\}$, and $\mathcal{C}_3 \in 2^{\Sigma_2}$. Then:*

$$\frac{\begin{array}{c}\mathcal{M}_1\langle\Sigma_{A_1^{safe}}\rangle, R_1 \models^{prt} A_1^{safe} \to A_2^{safe} \\ \mathcal{M}_2\langle\Sigma_{A_2^{safe}}\rangle, R_2 \models^{prt} A_2^{safe} \to G^{safe} \\ \mathcal{M}_2, R_3 \models^{cmp} A_1^{safe}\end{array}}{\mathcal{M}_1 \parallel \mathcal{M}_2, R_1 \cap R_2 \cap R_3 \models^{cmp} G^{safe}} \qquad \frac{\begin{array}{c}\mathcal{M}_1\langle\Sigma_{A_1}\rangle, R_1 \models^{fair_{\mathcal{C}_1}} A_1 \to A_2 \\ \mathcal{M}_2\langle\Sigma_{A_2}\rangle, R_2 \models^{fair_{\mathcal{C}_2}} A_2 \to G \\ \mathcal{M}_2, R_3 \models^{fair_{\mathcal{C}_3}} A_1\end{array}}{\mathcal{M}_1 \parallel \mathcal{M}_2, R_1 \cap R_2 \cap R_3 \models^{fair_{\mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3}} G}$$

Proof sketch. Similar to Theorem 33, the proof of the circular rules makes use of Theorem 30, which establishes that the composition under a strategy satisfies the same properties as the individual components under their corresponding projections. For safety, Lemma 25 allows us to verify the condition for complete strategies rather than partial strategies in the third premise. For fairness, Lemma 14 ensures that strategy projections remain fair. ◀

▶ Remark 36. The inclusion of fairness in the premises of the right rules in Theorem 33 and Theorem 35 enables recursive application and thus supports the compositional verification of systems with more than two components. In the case of a single application of one of the rules, it is sufficient to verify with respect to complete strategies, which, while a stronger condition, simplifies the verification process.

6 Compositional Reasoning about Monotonicity

Exploiting monotonicity can significantly enhance the efficiency of parameter synthesis [51]. However, determining monotonicity is computationally hard[5] and it would be beneficial to determine monotonicity in a compositional way. Additionally, monotonicity in composed pPA is challenging due to the complexities introduced by parameter dependencies and interactions among components. While we focus on global monotonicity, the following results can be extended to *local monotonicity*, which considers only the first transition from a given state. See [49, Definitions 4.4 and 4.5].

The probability of a language or the expected total reward for a pPA $\mathcal{M}$ can be viewed as a function – called *solution function* – that maps a well-defined parameter valuation to the corresponding probability or expected total reward, respectively [27, Definition 4.7].

▶ **Definition 37** (Solution Function). *Let $\mathcal{M}$ be a pPA over Σ, let $\sigma \in Str_{\mathcal{M}}$ and let R be a well-defined region. The* solution function *for $\mathcal{M}$ and language $\mathcal{L} \subseteq \Sigma^\infty$ is $\mathsf{sol}^{Pr(\mathcal{L})}_{\mathcal{M},\sigma} : R \to [0,1]$, where $\mathsf{sol}^{Pr(\mathcal{L})}_{\mathcal{M},\sigma}(v) = Pr^{v,\sigma}_{\mathcal{M}}(\mathcal{L})$. The solution function for $\mathcal{M}$ and a reward function $\mathcal{R}$ over Σ is $\mathsf{sol}^{Ex(\mathcal{R})}_{\mathcal{M},\sigma} : R \to \mathbb{R}_{\geq 0}$, where $\mathsf{sol}^{Ex(\mathcal{R})}_{\mathcal{M},\sigma}(v) = Ex^{v,\sigma}_{\mathcal{M}}(\mathcal{R})$.*

When referring to a solution function without specifying whether it pertains to probabilities or expected rewards, we simply write $\mathsf{sol}_{\mathcal{M},\sigma}$.

▶ **Example 38** (Solution Function). Consider the pPA $\mathcal{M}_1 \parallel \mathcal{M}_2$ in Figure 2 and the region $R = \{v\colon \{p,q\} \to [0,1]\}$ which is well-defined for $\mathcal{M}_1 \parallel \mathcal{M}_2$. Let $\mathcal{L} = \{w \in \{\mathsf{a},\mathsf{c},☹\}^\infty \mid |w|_{☹} = 0\}$ be the language of words over $\{\mathsf{a},\mathsf{b},\mathsf{c},☹\}$ that do not contain ☹. Let σ be the complete strategy of $\mathcal{M}_1 \parallel \mathcal{M}_2$ from Example 17, which always selects action a,c or ☹ with probability 1 whenever any of them is enabled; otherwise, it chooses b with probability 1. The solution function $\mathsf{sol}^{Pr(\mathcal{L})}_{\mathcal{M}_1\parallel\mathcal{M}_2,\sigma} : R \to [0,1]$ is defined by $\mathsf{sol}^{Pr(\mathcal{L})}_{\mathcal{M}_1\parallel\mathcal{M}_2,\sigma}(p,q) = 1 - \big(p^2 \cdot \frac{1}{10} + p\cdot(1-p)\cdot(p\cdot q + (1-p)\cdot q)\big) = 1 - \big(p^2\cdot\frac{1}{10} + (p-p^2)\cdot q\big)$.

We extend the standard notion of monotonicity [49] by differentiating between different strategy classes, including complete, partial, and fair strategies.

▶ **Definition 39** (Monotonicity). *Let σ be a strategy of $\mathcal{M}$. A solution function $\mathsf{sol}_{\mathcal{M},\sigma}$ is* monotonic increasing *in $p \in V$ on region R – denoted $\mathsf{sol}_{\mathcal{M},\sigma}{\uparrow}_{p,R}$ – if for all $v, v_+ \in R$ with $v_+(q) = v(q) + x \cdot [\![p{=}q]\!]$ for $q \in V$ and some $x \geq 0$, we have: $\mathsf{sol}_{\mathcal{M},\sigma}(v) \leq \mathsf{sol}_{\mathcal{M},\sigma}(v_+)$.*

For $\star \in \{prt, cmp\}$, we write $\mathsf{sol}_{\mathcal{M}}{\uparrow}^{\star}_{p,R}$ if $\mathsf{sol}_{\mathcal{M},\sigma}{\uparrow}_{p,R}$ for all $\sigma \in Str^{\star}_{\mathcal{M}}$. If R is graph-preserving, we write $\mathsf{sol}_{\mathcal{M}}{\uparrow}^{fair_c}_{p,R}$ if $\mathsf{sol}_{\mathcal{M},\sigma}{\uparrow}_{p,R}$ holds for all fair strategies $\sigma \in Str^{fair_c}_{\mathcal{M}[v]}$, $v \in R$. Notations $\mathsf{sol}_{\mathcal{M},\sigma}{\downarrow}_{p,R}$ and $\mathsf{sol}_{\mathcal{M}}{\downarrow}^{\star}_{p,R}$ for monotonic decreasing *$\mathsf{sol}_{\mathcal{M},\sigma}$ are defined analogously.*

We require the region to be graph-preserving when defining monotonicity w.r.t. fair strategies. This ensures that for any two valuations, v, v_+, we have $Str^{fair_c}_{\mathcal{M}[v]} = Str^{fair_c}_{\mathcal{M}[v_+]}$; see Proposition 10.

▶ Remark 40. Monotonicity for partial strategies w.r.t. general properties is equivalent to monotonicity for complete strategies in a modified pPA; see [38].

The following theorem states that monotonicity of a composed system can be verified by analyzing its individual components.

[5] For deterministic pPA (Markov chains) determining monotonicity is coETR-hard [49, Sec. 3.4].

▶ **Theorem 41** (Monotonicity). *Let $\mathcal{M}_1$, $\mathcal{M}_2$ be pPAs with alphabets Σ_1 and Σ_2 and R_i be a graph-preserving region for $\mathcal{M}_i$. Let $\mathsf{sol} \in \{\mathsf{sol}^{Pr(\mathcal{L})}, \mathsf{sol}^{Ex(\mathcal{R})}\}$ be a solution function w.r.t. the language $\mathcal{L}$ or reward function $\mathcal{R}$ over $\Sigma \subseteq (\Sigma_1 \cup \Sigma_2)$ and let $\updownarrow \in \{\uparrow, \downarrow\}$. Let $\mathcal{C}_i \subseteq 2^{\Sigma_1 \cup \Sigma}$. Then the following two proof rules hold:*

$$\frac{\mathsf{sol}_{\mathcal{M}_1\langle\Sigma\rangle}\updownarrow^{prt}_{p,R_1} \quad \mathsf{sol}_{\mathcal{M}_2\langle\Sigma\rangle}\updownarrow^{prt}_{p,R_2}}{\mathsf{sol}_{\mathcal{M}_1\|\mathcal{M}_2}\updownarrow^{prt}_{p,R_1\cap R_2}} \qquad \frac{\mathsf{sol}_{\mathcal{M}_1\langle\Sigma\rangle}\updownarrow^{fair_{\mathcal{C}_1}}_{p,R_1} \quad \mathsf{sol}_{\mathcal{M}_2\langle\Sigma\rangle}\updownarrow^{fair_{\mathcal{C}_2}}_{p,R_2}}{\mathsf{sol}_{\mathcal{M}_1\|\mathcal{M}_2}\updownarrow^{fair_{\mathcal{C}_1\cup\mathcal{C}_2}}_{p,R_1\cap R_2}}$$

Proof. We show the premises imply $\mathsf{sol}_{\mathcal{M}_1\langle\Sigma\rangle\|\mathcal{M}_2\langle\Sigma\rangle}\updownarrow^{\star}_{p,R_1\cap R_2}$ for $\star \in \{prt, fair_{\mathcal{C}_1\cup\mathcal{C}_2}\}$, which directly implies that $\mathsf{sol}_{\mathcal{M}_1\|\mathcal{M}_2}\updownarrow^{\star}_{p,R_1\cap R_2}$ holds, see Remark 31. We focus on the left rule, i.e., $\star = prt$.

▶ Remark 42. The proof for $\star = fair_{\mathcal{C}_1\cup\mathcal{C}_2}$ is similar but additionally requires Lemma 14 in [38, Appendix A.4].

We further consider $\updownarrow = \uparrow$. The case $\updownarrow = \downarrow$ follows analogously. Our proof is by contradiction. Assume that the premises hold but $\mathsf{sol}_{\mathcal{M}_1\langle\Sigma\rangle\|\mathcal{M}_2\langle\Sigma\rangle}\uparrow^{\star}_{p,R_1\cap R_2}$ does not hold. Thus, there is a strategy $\sigma \in Str^{\star}_{(\mathcal{M}_1\langle\Sigma\rangle\|\mathcal{M}_2\langle\Sigma\rangle)}$ and valuations $v, v_+ \in R_1 \cap R_2$ with $v_+(q) = v(q) + x \cdot [\![p{=}q]\!]$ for $q \in V$ and some $x \geq 0$ and

$$\mathsf{sol}_{(\mathcal{M}_1\langle\Sigma\rangle\|\mathcal{M}_2\langle\Sigma\rangle),\sigma}(v) > \mathsf{sol}_{(\mathcal{M}_1\langle\Sigma\rangle\|\mathcal{M}_2\langle\Sigma\rangle),\sigma}(v_+). \tag{1}$$

Theorem 30 yields $\mathsf{sol}_{(\mathcal{M}_1\langle\Sigma\rangle\|\mathcal{M}_2\langle\Sigma\rangle),\sigma}(v) = \mathsf{sol}_{\mathcal{M}_1\langle\Sigma\rangle,\sigma\restriction_1^{v,v}}(v)$. We have $\mathbf{P}_{\mathcal{M}_1\langle\Sigma\rangle}(s,\alpha,s')[v] = 0$ iff $\mathbf{P}_{\mathcal{M}_1\langle\Sigma\rangle}(s,\alpha,s')[v_+] = 0$ as v and v_+ are graph preserving for $\mathcal{M}_1$. Thus, we can apply Lemma 18 and obtain

$$\begin{aligned}
\mathsf{sol}_{\mathcal{M}_1\langle\Sigma\rangle,\sigma\restriction_1^{v,v}}(v) &= \mathsf{sol}_{\mathcal{M}_1\langle\Sigma\rangle,\sigma\restriction_1^{v_+,v}}(v) && \text{(by Lemma 18)}\\
&\leq \mathsf{sol}_{\mathcal{M}_1\langle\Sigma\rangle,\sigma\restriction_1^{v_+,v}}(v_+) && (\ \sigma\restriction_1^{v_+,v} \in Str^{prt}_{\mathcal{M}_1\langle\Sigma\rangle},\ \mathsf{sol}_{\mathcal{M}_1\langle\Sigma\rangle}\uparrow^{prt}_{p,R_1})\\
&= \mathsf{sol}_{(\mathcal{M}_1\langle\Sigma\rangle\|\mathcal{M}_2\langle\Sigma\rangle[v]),\sigma}(v_+) && \text{(by Theorem 30)}
\end{aligned}$$

We observe that

$$\big(\mathcal{M}_1\langle\Sigma\rangle \parallel \mathcal{M}_2\langle\Sigma\rangle[v]\big)[v_+] \;=\; \big(\mathcal{M}_1\langle\Sigma\rangle[v_+] \parallel \mathcal{M}_2\langle\Sigma\rangle[v]\big) \;=\; \big(\mathcal{M}_1\langle\Sigma\rangle[v_+] \parallel \mathcal{M}_2\langle\Sigma\rangle\big)[v].$$

Consequently, $\mathsf{sol}_{(\mathcal{M}_1\langle\Sigma\rangle\|\mathcal{M}_2\langle\Sigma\rangle[v]),\sigma}(v_+) = \mathsf{sol}_{(\mathcal{M}_1\langle\Sigma\rangle[v_+]\|\mathcal{M}_2\langle\Sigma\rangle,\sigma}(v)$. By a similar reasoning as above, we obtain

$$\begin{aligned}
\mathsf{sol}_{(\mathcal{M}_1\langle\Sigma\rangle[v_+]\|\mathcal{M}_2\langle\Sigma\rangle),\sigma}(v) &= \mathsf{sol}_{\mathcal{M}_2\langle\Sigma\rangle,\sigma\restriction_2^{v_+,v}}(v) && \text{(by Theorem 30)}\\
&\leq \mathsf{sol}_{\mathcal{M}_2\langle\Sigma\rangle,\sigma\restriction_2^{v_+,v}}(v_+) && (\ \sigma\restriction_2^{v_+,v} \in Str^{prt}_{\mathcal{M}_2\langle\Sigma\rangle},\ \mathsf{sol}_{\mathcal{M}_2\langle\Sigma\rangle}\uparrow^{prt}_{p,R_2})\\
&= \mathsf{sol}_{\mathcal{M}_2\langle\Sigma\rangle,\sigma\restriction_2^{v_+,v_+}}(v_+) && \text{(by Lemma 18)}\\
&= \mathsf{sol}_{(\mathcal{M}_1\langle\Sigma\rangle\|\mathcal{M}_2\langle\Sigma\rangle),\sigma}(v_+) && \text{(by Theorem 30)}
\end{aligned}$$

Thus, $\mathsf{sol}_{(\mathcal{M}_1\langle\Sigma\rangle\|\mathcal{M}_2\langle\Sigma\rangle),\sigma}(v) \leq \mathsf{sol}_{(\mathcal{M}_1\langle\Sigma\rangle\|\mathcal{M}_2\langle\Sigma\rangle),\sigma}(v_+)$, violating Equation (1). ◀

▶ **Example 43.** Reconsider the pPA $\mathcal{M}_1 \parallel \mathcal{M}_2$ in Figure 2, the region $R = \{v\colon \{p,q\} \to [0,1]\}$, and the language $\mathcal{L} = \{w \in \{a,c,☹\}^{\infty} \mid |w|_{☹} = 0\}$ from Example 38. The pPA $\mathcal{M}_1 \parallel \mathcal{M}_2$ is composed of the pPAs $\mathcal{M}_1$ and $\mathcal{M}_2$ shown in Figure 1. The region R is well-defined for $\mathcal{M}_1$ and $\mathcal{M}_2$. We check whether $\mathsf{sol}^{Pr(\mathcal{L})}_{\mathcal{M}_1\|\mathcal{M}_2}$ is monotonic in q on R via Theorem 41. Since the premises $\mathsf{sol}^{Pr(\mathcal{L})}_{\mathcal{M}_i\langle\Sigma_{\mathcal{L}}\rangle}\downarrow^{prt}_{q,R}$ for $i \in \{1,2\}$ are satisfied, we conclude $\mathsf{sol}^{Pr(\mathcal{L})}_{\mathcal{M}_1\|\mathcal{M}_2}\downarrow^{prt}_{q,R}$.

7 Related Work

Compositional verification has been widely studied in both probabilistic and non-probabilistic systems. We summarize key contributions related to our work.

Jones' rely-guarantee method [26] and Pnueli's compositional proof system [44] for temporal logic laid the foundation for AG reasoning. Subsequent work focused on AG rules for systems using CTL* [12] and developed AG reasoning for *reactive modules* [25, 1]. Automated AG reasoning techniques – developed by Pasareanu et al. [13, 41] – include learning-based assumption generation. More recent work has focused on circular AG reasoning [16] and verification-repair approaches [22].

AG reasoning has been lifted to probabilistic settings. Initial work by de Alfaro et al. [15] introduced AG rules for a probabilistic extension of reactive modules [25, 1]. Their model is similar to PA [47, 46], but limited to synchronous composition.

Kwiatkowska et al. [34, 21] generalized AG verification for PA, allowing more flexible parallel compositions and extending AG reasoning to probabilistic safety properties. Their approach reduces AG verification to multi-objective model checking, as proposed by Etessami et al. [17]. This was further refined in [35], enabling AG reasoning over a broader class of quantitative properties, including conjunctions over probabilistic liveness and expected rewards. Algorithmic learning-based assumption generation techniques [13, 23] were later adapted for AG reasoning in probabilistic settings [18, 19, 36]. Other assumption generation approaches include abstraction-refinement methods [32, 10], based on the CEGAR paradigm [11], and symbolic learning-based methods [24, 5]. AG reasoning has been applied to various real-world domains, including service-based workflow verification [6], large-scale IT systems [7], and autonomous systems with deep neural networks [42, 43].

AG reasoning has also been extended to systems with uncertainty, for example, [55] introduced an AG framework for verifying systems with components modeled by MDPs and *partially observable MDPs* (POMDPs). In contrast, our work considers a different type of uncertainty; We extend AG reasoning to parametric probabilistic automata (PA), leveraging research on parametric MDPs [27, 45, 28] and previous AG verification techniques [35]. Our framework allows to reason about monotonicity [49, 50, 51] in a compositional manner. To the best of our knowledge, this the first AG-based compositional verification framework for parametric PA. Existing modular proof systems have focused on parametric timed automata [2] or non-probabilistic parameterized systems [4, 48, 39], where concurrent programs are parameterized by the number of processes or threads in a configured instance.

Another recent line of research focuses on the *sequential* composition of MDPs rather than parallel decomposition: Junges and Spaan [30] introduced an abstraction-refinement approach for hierarchical probabilistic models, leveraging parametric MDPs to represent sets of similar subroutines. Recent work by Watanabe et al. [53] on mean-payoff games, applies category-theoretic string diagrams to the verification of sequentially composed MDPs.

8 Conclusion

We presented an assume-guarantee framework for compositional verification of parametric probabilistic automata, building on the proof rules for Segala's PA by Kwiatkowska et al. [35]. In addition, we introduced new compositional proof rules to reason about monotonicity in composed systems. These contributions lay the theoretical foundations for modular verification of pPA. To the best of our knowledge, these are the first AG proof rules for probabilistic models with parametric transition probabilities.

Future work involves implementing the framework and demonstrating its effectiveness through case studies. Another direction is to deduce additional assume-guarantee rules – for example, reasoning about robust valuations or strategies, i.e, properties of the form: $\exists v \in R : \forall \sigma \in Str^{\star}_{\mathcal{M}} : \mathcal{M}[v], \sigma \models \mathsf{X}$. Additionally, interesting directions include the modular verification of other properties, such as long-run average rewards or expected visiting times [37]. Other areas include extending verification to logics such as parametric LTL [9] and probabilistic CTL. Further research could also explore Markov automata with parameters, building on preliminary work in modular reasoning for continuous-time and continuous-space models [8]. Another interesting direction is adapting assume-guarantee reasoning for stochastic games [54] to a parametric setting.

References

1 Rajeev Alur and Thomas A. Henzinger. Reactive modules. *Formal Methods Syst. Des.*, 15(1):7–48, 1999. doi:10.1023/A:1008739929481.

2 Lacramioara Astefanoaei, Saddek Bensalem, Marius Bozga, Chih-Hong Cheng, and Harald Ruess. Compositional parameter synthesis. In John S. Fitzgerald, Constance L. Heitmeyer, Stefania Gnesi, and Anna Philippou, editors, *FM 2016: Formal Methods - 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings*, volume 9995 of *Lecture Notes in Computer Science*, pages 60–68, 2016. doi:10.1007/978-3-319-48989-6_4.

3 Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.

4 Samik Basu and C. R. Ramakrishnan. Compositional analysis for verification of parameterized systems. *Theor. Comput. Sci.*, 354(2):211–229, 2006. doi:10.1016/J.TCS.2005.11.016.

5 Redouane Bouchekir and Mohand Cherif Boukala. Toward implicit learning for the compositional verification of Markov decision processes. In Mohamed Faouzi Atig, Saddek Bensalem, Simon Bliudze, and Bruno Monsuez, editors, *Verification and Evaluation of Computer and Communication Systems - 12th International Conference, VECoS 2018, Grenoble, France, September 26-28, 2018, Proceedings*, volume 11181 of *Lecture Notes in Computer Science*, pages 200–217. Springer, 2018. doi:10.1007/978-3-030-00359-3_13.

6 Redouane Bouchekir, Saïda Boukhedouma, and Mohand Cherif Boukala. Automatic compositional verification of probabilistic safety properties for inter-organisational workflow processes. In Yuri Merkuryev, Tuncer I. Ören, and Mohammad S. Obaidat, editors, *Proceedings of the 6th International Conference on Simulation and Modeling Methodologies, Technologies and Applications (SIMULTECH 2016), Lisbon, Portugal, July 29-31, 2016*, pages 244–253. SciTePress, 2016. doi:10.5220/0005978602440253.

7 Radu Calinescu, Shinji Kikuchi, and Kenneth Johnson. Compositional reverification of probabilistic safety properties for large-scale complex IT systems. In Radu Calinescu and David Garlan, editors, *Large-Scale Complex IT Systems. Development, Operation and Management - 17th Monterey Workshop 2012, Oxford, UK, March 19-21, 2012, Revised Selected Papers*, volume 7539 of *Lecture Notes in Computer Science*, pages 303–329. Springer, 2012. doi:10.1007/978-3-642-34059-8_16.

8 Luca Cardelli, Kim G. Larsen, and Radu Mardare. Modular Markovian logic. In Luca Aceto, Monika Henzinger, and Jirí Sgall, editors, *Automata, Languages and Programming - 38th International Colloquium, ICALP 2011, Zurich, Switzerland, July 4-8, 2011, Proceedings, Part II*, volume 6756 of *Lecture Notes in Computer Science*, pages 380–391. Springer, 2011. doi:10.1007/978-3-642-22012-8_30.

9 Souymodip Chakraborty and Joost-Pieter Katoen. Parametric LTL on Markov chains. In Josep Díaz, Ivan Lanese, and Davide Sangiorgi, editors, *Theoretical Computer Science - 8th IFIP TC 1/WG 2.2 International Conference, TCS 2014, Rome, Italy, September 1-3, 2014. Proceedings*, volume 8705 of *Lecture Notes in Computer Science*, pages 207–221. Springer, 2014. doi:10.1007/978-3-662-44602-7_17.

10 Krishnendu Chatterjee, Martin Chmelik, and Przemyslaw Daca. CEGAR for compositional analysis of qualitative properties in Markov decision processes. *Formal Methods Syst. Des.*, 47(2):230–264, 2015. doi:10.1007/S10703-015-0235-2.

11 Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. Allen Emerson and A. Prasad Sistla, editors, *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000. doi:10.1007/10722167_15.

12 Edmund M. Clarke, David E. Long, and Kenneth L. McMillan. Compositional model checking. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989*, pages 353–362. IEEE Computer Society, 1989. doi:10.1109/LICS.1989.39190.

13 Jamieson M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Pasareanu. Learning assumptions for compositional verification. In Hubert Garavel and John Hatcliff, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference, TACAS 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*, volume 2619 of *Lecture Notes in Computer Science*, pages 331–346. Springer, 2003. doi:10.1007/3-540-36577-X_24.

14 Conrado Daws. Symbolic and parametric model checking of discrete-time Markov chains. In *ICTAC*, volume 3407 of *Lecture Notes in Computer Science*, pages 280–294. Springer, 2004. doi:10.1007/978-3-540-31862-0_21.

15 Luca de Alfaro, Thomas A. Henzinger, and Ranjit Jhala. Compositional methods for probabilistic systems. In Kim Guldstrand Larsen and Mogens Nielsen, editors, *CONCUR 2001 - Concurrency Theory, 12th International Conference, Aalborg, Denmark, August 20-25, 2001, Proceedings*, volume 2154 of *Lecture Notes in Computer Science*, pages 351–365. Springer, 2001. doi:10.1007/3-540-44685-0_24.

16 Karam Abd Elkader, Orna Grumberg, Corina S. Pasareanu, and Sharon Shoham. Automated circular assume-guarantee reasoning. *Formal Aspects Comput.*, 30(5):571–595, 2018. doi:10.1007/S00165-017-0436-0.

17 Kousha Etessami, Marta Z. Kwiatkowska, Moshe Y. Vardi, and Mihalis Yannakakis. Multi-objective model checking of Markov decision processes. *Log. Methods Comput. Sci.*, 4(4), 2008. doi:10.2168/LMCS-4(4:8)2008.

18 Lu Feng, Marta Z. Kwiatkowska, and David Parker. Compositional verification of probabilistic systems using learning. In *QEST 2010, Seventh International Conference on the Quantitative Evaluation of Systems, Williamsburg, Virginia, USA, 15-18 September 2010*, pages 133–142. IEEE Computer Society, 2010. doi:10.1109/QEST.2010.24.

19 Lu Feng, Marta Z. Kwiatkowska, and David Parker. Automated learning of probabilistic assumptions for compositional reasoning. In Dimitra Giannakopoulou and Fernando Orejas, editors, *Fundamental Approaches to Software Engineering - 14th International Conference, FASE 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*, volume 6603 of *Lecture Notes in Computer Science*, pages 2–17. Springer, 2011. doi:10.1007/978-3-642-19811-3_2.

20 Lu Feng, Clemens Wiltsche, Laura R. Humphrey, and Ufuk Topcu. Controller synthesis for autonomous systems interacting with human operators. In *ICCPS*, pages 70–79. ACM, 2015. doi:10.1145/2735960.2735973.

21 Vojtech Forejt, Marta Z. Kwiatkowska, Gethin Norman, and David Parker. Automated verification techniques for probabilistic systems. In Marco Bernardo and Valérie Issarny, editors, *Formal Methods for Eternal Networked Software Systems - 11th International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM 2011, Bertinoro, Italy, June 13-18, 2011. Advanced Lectures*, volume 6659 of *Lecture Notes in Computer Science*, pages 53–113. Springer, 2011. doi:10.1007/978-3-642-21455-4_3.

22 Hadar Frenkel, Orna Grumberg, Corina S. Pasareanu, and Sarai Sheinvald. Assume, guarantee or repair: a regular framework for non regular properties. *Int. J. Softw. Tools Technol. Transf.*, 24(5):667–689, 2022. doi:10.1007/S10009-022-00669-9.

23 Anubhav Gupta, Kenneth L. McMillan, and Zhaohui Fu. Automated assumption generation for compositional verification. *Formal Methods Syst. Des.*, 32(3):285–301, 2008. doi:10.1007/S10703-008-0050-0.

24 Fei He, Xiaowei Gao, Miaofei Wang, Bow-Yaw Wang, and Lijun Zhang. Learning weighted assumptions for compositional verification of Markov decision processes. *ACM Trans. Softw. Eng. Methodol.*, 25(3):21:1–21:39, 2016. doi:10.1145/2907943.

25 Thomas A. Henzinger, Shaz Qadeer, and Sriram K. Rajamani. You assume, we guarantee: Methodology and case studies. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer Aided Verification, 10th International Conference, CAV '98, Vancouver, BC, Canada, June 28 - July 2, 1998, Proceedings*, volume 1427 of *Lecture Notes in Computer Science*, pages 440–451. Springer, 1998. doi:10.1007/BFB0028765.

26 Cliff B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, 1983. doi:10.1145/69575.69577.

27 Sebastian Junges. *Parameter synthesis in Markov models*. PhD thesis, RWTH Aachen University, Germany, 2020. URL: https://publications.rwth-aachen.de/record/783179.

28 Sebastian Junges, Erika Ábrahám, Christian Hensel, Nils Jansen, Joost-Pieter Katoen, Tim Quatmann, and Matthias Volk. Parameter synthesis for Markov models: covering the parameter space. *Formal Methods Syst. Des.*, 62(1):181–259, 2024. doi:10.1007/S10703-023-00442-X.

29 Sebastian Junges, Joost-Pieter Katoen, Guillermo A. Pérez, and Tobias Winkler. The complexity of reachability in parametric Markov decision processes. *J. Comput. Syst. Sci.*, 119:183–210, 2021. doi:10.1016/J.JCSS.2021.02.006.

30 Sebastian Junges and Matthijs T. J. Spaan. Abstraction-refinement for hierarchical probabilistic models. In Sharon Shoham and Yakir Vizel, editors, *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part I*, volume 13371 of *Lecture Notes in Computer Science*, pages 102–123. Springer, 2022. doi:10.1007/978-3-031-13185-1_6.

31 Joost-Pieter Katoen. The probabilistic model checking landscape. In *LICS*, pages 31–45. ACM, 2016. doi:10.1145/2933575.2934574.

32 Anvesh Komuravelli, Corina S. Pasareanu, and Edmund M. Clarke. Assume-guarantee abstraction refinement for probabilistic systems. In P. Madhusudan and Sanjit A. Seshia, editors, *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, volume 7358 of *Lecture Notes in Computer Science*, pages 310–326. Springer, 2012. doi:10.1007/978-3-642-31424-7_25.

33 Marta Z. Kwiatkowska, Gethin Norman, and David Parker. Using probabilistic model checking in systems biology. *SIGMETRICS Perform. Evaluation Rev.*, 35(4):14–21, 2008. doi:10.1145/1364644.1364651.

34 Marta Z. Kwiatkowska, Gethin Norman, David Parker, and Hongyang Qu. Assume-guarantee verification for probabilistic systems. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6015 of *Lecture Notes in Computer Science*, pages 23–37. Springer, 2010. doi:10.1007/978-3-642-12002-2_3.

35 Marta Z. Kwiatkowska, Gethin Norman, David Parker, and Hongyang Qu. Compositional probabilistic verification through multi-objective model checking. *Inf. Comput.*, 232:38–65, 2013. doi:10.1016/J.IC.2013.10.001.

36 Rui Li and Yang Liu. Compositional stochastic model checking probabilistic automata via symmetric assume-guarantee rule. In *17th IEEE International Conference on Software Engineering Research, Management and Applications, SERA 2019, Honolulu, HI, USA, May 29-31, 2019*, pages 110–115. IEEE, 2019. doi:10.1109/SERA.2019.8886808.

37 Hannah Mertens, Joost-Pieter Katoen, Tim Quatmann, and Tobias Winkler. Accurately computing expected visiting times and stationary distributions in Markov chains. In Bernd Finkbeiner and Laura Kovács, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 30th International Conference, TACAS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part II*, volume 14571 of *Lecture Notes in Computer Science*, pages 237–257. Springer, 2024. doi:10.1007/978-3-031-57249-4_12.

38 Hannah Mertens, Tim Quatmann, and Joost-Pieter Katoen. Compositional reasoning for parametric probabilistic automata, 2025. arXiv:2506.08525.

39 Kedar S. Namjoshi and Richard J. Trefler. Parameterized compositional model checking. In Marsha Chechik and Jean-François Raskin, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, volume 9636 of *Lecture Notes in Computer Science*, pages 589–606. Springer, 2016. doi:10.1007/978-3-662-49674-9_39.

40 Gethin Norman and Vitaly Shmatikov. Analysis of probabilistic contract signing. *J. Comput. Secur.*, 14(6):561–589, 2006. doi:10.3233/jcs-2006-14604.

41 Corina S. Pasareanu, Dimitra Giannakopoulou, Mihaela Gheorghiu Bobaru, Jamieson M. Cobleigh, and Howard Barringer. Learning to divide and conquer: applying the L* algorithm to automate assume-guarantee reasoning. *Formal Methods Syst. Des.*, 32(3):175–205, 2008. doi:10.1007/S10703-008-0049-6.

42 Corina S. Pasareanu, Divya Gopinath, and Huafeng Yu. Compositional verification for autonomous systems with deep learning components. *CoRR*, abs/1810.08303, 2018. arXiv:1810.08303.

43 Corina S. Pasareanu, Ravi Mangal, Divya Gopinath, and Huafeng Yu. Assumption generation for learning-enabled autonomous systems. In Panagiotis Katsaros and Laura Nenzi, editors, *Runtime Verification - 23rd International Conference, RV 2023, Thessaloniki, Greece, October 3-6, 2023, Proceedings*, volume 14245 of *Lecture Notes in Computer Science*, pages 3–22. Springer, 2023. doi:10.1007/978-3-031-44267-4_1.

44 Amir Pnueli. In transition from global to modular temporal reasoning about programs. In Krzysztof R. Apt, editor, *Logics and Models of Concurrent Systems - Conference proceedings, Colle-sur-Loup (near Nice), France, 8-19 October 1984*, volume 13 of *NATO ASI Series*, pages 123–144. Springer, 1984. doi:10.1007/978-3-642-82453-1_5.

45 Tim Quatmann, Christian Dehnert, Nils Jansen, Sebastian Junges, and Joost-Pieter Katoen. Parameter synthesis for Markov models: Faster than ever. In Cyrille Artho, Axel Legay, and Doron Peled, editors, *Automated Technology for Verification and Analysis - 14th International Symposium, ATVA 2016, Chiba, Japan, October 17-20, 2016, Proceedings*, volume 9938 of *Lecture Notes in Computer Science*, pages 50–67, 2016. doi:10.1007/978-3-319-46520-3_4.

46 Roberto Segala. *Modeling and verification of randomized distributed real-time systems*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1995. URL: https://hdl.handle.net/1721.1/36560.

47 Roberto Segala and Nancy A. Lynch. Probabilistic simulations for probabilistic processes. *Nord. J. Comput.*, 2(2):250–273, 1995.

48 Antti Siirtola and Keijo Heljanko. Parametrised compositional verification with multiple process and data types. In Josep Carmona, Mihai T. Lazarescu, and Marta Pietkiewicz-Koutny, editors, *13th International Conference on Application of Concurrency to System Design, ACSD 2013, Barcelona, Spain, 8-10 July, 2013*, pages 60–69. IEEE Computer Society, 2013. doi:10.1109/ACSD.2013.9.

49 Jip Spel. *Monotonicity in Markov models*. PhD thesis, RWTH Aachen University, Germany, 2023. URL: https://publications.rwth-aachen.de/record/974903.

50 Jip Spel, Sebastian Junges, and Joost-Pieter Katoen. Are parametric Markov chains monotonic? In Yu-Fang Chen, Chih-Hong Cheng, and Javier Esparza, editors, *Automated Technology*

for Verification and Analysis - 17th International Symposium, ATVA 2019, Taipei, Taiwan, October 28-31, 2019, Proceedings, volume 11781 of *Lecture Notes in Computer Science*, pages 479–496. Springer, 2019. doi:10.1007/978-3-030-31784-3_28.

51 Jip Spel, Sebastian Junges, and Joost-Pieter Katoen. Finding provably optimal Markov chains. In Jan Friso Groote and Kim Guldstrand Larsen, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part I*, volume 12651 of *Lecture Notes in Computer Science*, pages 173–190. Springer, 2021. doi:10.1007/978-3-030-72016-2_10.

52 Mariëlle Stoelinga. An introduction to probabilistic automata. *Bull. EATCS*, 78:176–198, 2002.

53 Kazuki Watanabe, Clovis Eberhart, Kazuyuki Asada, and Ichiro Hasuo. Compositional solution of mean payoff games by string diagrams. In Nils Jansen, Sebastian Junges, Benjamin Lucien Kaminski, Christoph Matheja, Thomas Noll, Tim Quatmann, Mariëlle Stoelinga, and Matthias Volk, editors, *Principles of Verification: Cycling the Probabilistic Landscape - Essays Dedicated to Joost-Pieter Katoen on the Occasion of His 60th Birthday, Part III*, volume 15262 of *Lecture Notes in Computer Science*, pages 423–445. Springer, 2024. doi:10.1007/978-3-031-75778-5_20.

54 Clemens Wiltsche. *Assume-guarantee strategy synthesis for stochastic games*. PhD thesis, University of Oxford, UK, 2015. URL: https://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.719857.

55 Xiaobin Zhang, Bo Wu, and Hai Lin. Assume-guarantee reasoning framework for MDP-POMDP. In *55th IEEE Conference on Decision and Control, CDC 2016, Las Vegas, NV, USA, December 12-14, 2016*, pages 795–800. IEEE, 2016. doi:10.1109/CDC.2016.7798365.

Resolving Nondeterminism by Chance

Soumyajit Paul
University of Liverpool, UK

David Purser
University of Liverpool, UK

Sven Schewe
University of Liverpool, UK

Qiyi Tang
University of Liverpool, UK

Patrick Totzke
University of Liverpool, UK

Di-De Yen
University of Liverpool, UK

Abstract

History-deterministic automata are those in which nondeterministic choices can be correctly resolved stepwise: there is a strategy to select a continuation of a run given the next input letter so that if the overall input word admits some accepting run, then the constructed run is also accepting.

Motivated by checking qualitative properties in probabilistic verification, we consider the setting where the resolver strategy can randomise and only needs to succeed with lower-bounded probability. We study the expressiveness of such stochastically-resolvable automata as well as consider the decision questions of whether a given automaton has this property. In particular, we show that it is undecidable to check if a given NFA is λ-stochastically resolvable. This problem is decidable for finitely-ambiguous automata. We also present complexity upper and lower bounds for several well-studied classes of automata for which this problem remains decidable.

2012 ACM Subject Classification Theory of computation → Formal languages and automata theory

Keywords and phrases History-determinism, finite automata, probabilistic automata

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2025.32

Related Version *Full Version*: https://arxiv.org/abs/2504.10234

Funding This work has been supported by the European Union's Horizon Europe (HORIZON) programme under the Marie Skłodowska-Curie grant agreement No 101208673 and the EPSRC through the projects EP/X042596/1 and EP/X03688X/1.

Acknowledgements We would like to thank the anonymous reviewers for their constructive feedback, that helped improve the paper.

1 Introduction

Many successful verification techniques rely on automata representations for specifications that capture the languages of acceptable system traces. These automata models are typically nondeterministic: any given trace may give rise to multiple runs of the automaton, and is accepted if at least one of these runs is successful. This enables succinct representations but is also a major source of complexity due to costly intermediate determinisation steps.

It is therefore natural to put extra constraints on the extent to which an automaton allows nondeterministic choice to avoid determinisation. One way to do this is to bound the level of ambiguity: an automaton is k-ambiguous if on every accepted word it has at most k

© Soumyajit Paul, David Purser, Sven Schewe, Qiyi Tang, Patrick Totzke, and Di-De Yen;
licensed under Creative Commons License CC-BY 4.0

(a) NFA $\mathcal{A}$ is 1/2-resolvable. **(b)** NFA $\mathcal{B}$ is not positively resolvable.

Figure 1 Two unambiguous NFA (all missing transitions implicitly go to a non-accepting sink). The one on the left is λ-resolvable for all $\lambda \leq 1/2$. The one on the right is not positively resolvable, because no matter the choice of transition probability, the probability of b^n tends to zero as n grows.

many distinct accepting runs (see e.g. [42, 35, 16]). Unambiguous automata have proven to be useful for model-checking of Markov chain models [7, 33]. An orthogonal restriction on nondeterminism, which has been extensively studied in recent years (cf. [24, 14, 27, 3, 8, 1, 41, 9, 10, 32, 33]), is to demand that choices can be resolved "on-the-fly": An automaton is *history-deterministic* (HD) if there is a strategy to select a continuation of a run given a next letter, so that if the overall word admits some accepting run then the constructed run is also accepting. This condition is strict in the sense that the resolver strategy must guarantee to produce an accepting run if one exists. In some scenarios less strict guarantees may suffice, for example when model-checking Markov Decision Processes [22, 21, 4, 6]. This motivates the study of automata that can be resolved in a weaker sense, namely, where the resolver strategy can randomise and is only required to succeed with a lower-bounded confidence.

A *stochastic resolver* for some nondeterministic automaton $\mathcal{A}$ is a function that, for any finite run and input letter, gives a distribution over the possible transitions. This produces a probabilistic automaton $\mathcal{P}$ that assigns a probability of acceptance to every input word and, together with a threshold $\lambda > 0$, defines the language $\mathcal{L}(\mathcal{P}_{\geq\lambda})$ consisting of all words whose probability of acceptance is at least λ (see, e.g., [37]). Unless stated otherwise, we will consider *memoryless* stochastic resolvers, which base their decisions solely on the last state of the given run and the input letter. Fixing a memoryless resolver turns $\mathcal{A}$ into a probabilistic automaton $\mathcal{P}$ over the same set of states and where transitions that appear positively in $\mathcal{P}$ are also contained in $\mathcal{A}$. Consequently then, for every threshold $0 < \lambda$, the language $\mathcal{L}(\mathcal{P}_{\geq\lambda})$ is included in that of $\mathcal{A}$. We now ask which automata admit positional resolvers that also guarantee the inclusion in the other direction.

An automaton $\mathcal{A}$ called *λ-memoryless stochastically resolvable* (or simply λ-resolvable) if there exists a memoryless stochastic resolver with $\mathcal{L}(\mathcal{P}_{\geq\lambda}) = \mathcal{L}(\mathcal{A})$. An automaton is *positively memoryless stochastically resolvable* (or simply positively resolvable) if it is λ-resolvable for some $\lambda > 0$. By varying the threshold λ, stochastic resolvability defines a spectrum where 0-resolvable corresponds to unrestricted nondeterminism and, on finite words, 1-resolvable coincides with history-determinism. See Figure 1 for distinguishing examples.

Related Work. The notion of history-determinism was introduced independently, with slightly different definitions, by Henzinger and Piterman [24] for solving games without determinisation, by Colcombet [14] for cost-functions, and by Kupferman, Safra, and Vardi [28] for recognising derived tree languages of word automata. For ω-regular automata, these variations all coincide [8]. For coBüchi-recognisable languages, history-deterministic automata can be exponentially more succinct than any equivalent deterministic ones [27]. Checking whether a given automaton is HD is decidable in polynomial time for Büchi and coBüchi automata [3, 27] and more generally also for parity automata of any fixed parity index [31]. History-determinism has been studied for richer automata models, such as pushdown automata [32, 20] timed automata [12, 11, 23] and quantitative automata [9, 10].

Recently, and independently from us, Henzinger, Prakash and Thejaswini [25] introduced classes of *stochastically resolvable* automata. The main difference of their work compared to ours is that they study the qualitative setting, where resolvers are required to succeed *almost-surely*, with probability one. In the terminology introduced above these are 1-resolvable automata. They study such automata on infinite words and compare their expressiveness, relative succinctness, against history-deterministic and semantically deterministic automata. Over finite words, these notions coincide: an NFA is 1-stochastically resolvable iff it is 1-memoryless stochastically resolvable iff it is semantically deterministic iff it is history-deterministic. They also consider the complexity of determining whether an automaton is 1-resolvable and establish that this is in polynomial time for safety automata, and PSPACE-complete for reachability and weak automata, and remains open for more general classes.

For a fixed probabilistic automaton, deciding whether a given λ is a lower bound corresponds to the undecidable emptiness problem [37], while asking if such a λ exists relates to the undecidable zero-isolation problem [19]. These problems become more tractable under restrictions such as bounded ambiguity [15, 16, 17, 18]. In the unary case, checking if λ is a lower bound is equivalent to the long-standing open positivity problem for linear recurrence sequences [36] and is also hard for Markov chains [45]. In contrast, our setting allows the probabilistic automaton to vary, asking whether some choice of probabilities satisfies the desired properties.

The unary case can be seen as a synthesis problem on parametric Markov chains. Consider the distribution transformer perspective [2, 5], where a Markov chain induces a sequence of distributions over states. Then the λ-resolvability problem asks, for universal NFA, whether there exists a parameter assignment such that the sequence of distributions is "globally" in the semi-algebraic set $S = \{x \in [0,1]^Q \mid \sum_{q\in Q} x_q = 1 \text{ and } \sum_{q\in F} x_q \geq \lambda\}$ (distributions with probability mass at least λ in accepting states). When the NFA is non-universal the "globally" condition must be adjusted to the eventually periodic pattern of the NFA. Existing work characterises, up to a set of measure zero, parameters that satisfy prefix-independent properties or "eventually" properties such as Finally Globally in S [5]. For general parametric questions, there is always the special case where the parameters are redundant, which encodes the aforementioned positivity problem [45]. This special case does not necessarily arise in our setting, as the problem has *all* transitions parameterised independently (only the structure is fixed), which leaves hope for λ-resolvability.

Our Contributions. We focus on automata over finite words and consider the decision problems whether a given automaton is resolvable. We distinguish the two variants of this problem, asking whether the automaton is positively resolvable, or whether it is λ-resolvable for a given value of λ. Our results are as follows, see also Table 1 for a summary.

- We introduce the quantitative notion of λ-memoryless stochastic resolvability, and show λ-resolvability induces a strict hierarchy of automata with varying parameter $\lambda \in (0,1)$.
- We show that checking λ-resolvability is undecidable already for NFA, on finite words, and therefore also for automata on infinite words regardless of the accepting condition.
- We complement this by showing that both positive resolvability and λ-resolvability remain decidable for finitely-ambiguous automata.
- We present complexity upper and lower bounds for several well-studied variations of finitely-ambiguous automata (summarised in Table 1). In particular, checking positive resolvability is PSPACE-complete for finitely-ambiguous automata, NL-complete for unambiguous (1-ambiguous) automata, and in the polynomial hierarchy for (unrestricted) unary automata. Checking λ-resolvability is PSPACE-hard even for k-ambiguous automata (and coNP-hard over a unary alphabet).

Table 1 Deciding positive resolvability (PR) and λ-resolvability (λR) of NFAs where $\lambda \in (0, 1)$.

		unambiguous	finitely-ambiguous	general
PR	unary	NL (Thm. 8)	coNP-hard (Thm. 11) Σ_2^P (Thm. 4)	
	non-unary	NL-complete (Thms. 9 and 8)	PSPACE-complete (Thms. 12 and 19)	open
λR	unary	PTIME (Thm. 10)	coNP-hard (Thm. 11) decidable (Thm. 20)	open
	non-unary	NL-hard (Thm. 9) PTIME (Thm. 10)	PSPACE-hard (Thm. 12) decidable (Thm. 20)	undecidable (Thm. 2)

2 Preliminaries

In this paper, we use $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Z}_+$, $\mathbb{Q}$, $\mathbb{Q}_+$, and $\mathbb{R}$ to denote the sets of non-negative integers, integers, positive integers, rational numbers, positive rational numbers, and real numbers, respectively. For $S \in \{\mathbb{Z}, \mathbb{Q}, \mathbb{R}\}$ and two numbers $i, j \in S$ with $i < j$, the notation $[i, j]_S$ (resp., $(i, j)_S$) denotes the set $\{d \in S \mid i \leq d \leq j\}$ (resp., $\{d \in S \mid i < d < j\}$). The half-open intervals $(i, j]_S$ and $[i, j)_S$ are defined analogously. We also write $[j]_S$ to denote $[1, j]_S$. The subscript S is omitted if it is clear from the context.

Nondeterministic Finite Automata. A *nondeterministic finite automaton* (NFA) $\mathcal{A}$ consists of a finite alphabet Σ, a finite set of states Q, an initial state q_0, a set of accepting states $F \subseteq Q$, and a set of transitions $\Delta \subseteq Q \times \Sigma \times Q$. We also use $p \xrightarrow{\sigma} q$ to signify that $(p, \sigma, q) \in \Delta$. $\mathcal{A}$ is *unary* if $|\Sigma| = 1$; a *deterministic finite automaton* (DFA) if, for every $(p, \sigma) \in Q \times \Sigma$, there exists at most one state $q \in Q$ such that $p \xrightarrow{\sigma} q$; and *complete* if, for every $(p, \sigma) \in Q \times \Sigma$, there exists some $q \in Q$ such that $p \xrightarrow{\sigma} q$.

Given a word $w = \sigma_1 \dots \sigma_n \in \Sigma^*$, a *run* π of $\mathcal{A}$ on w is a sequence of transitions $\tau_1 \dots \tau_n$, where each transition $\tau_i = (p_i, \sigma_i, q_i) \in \Delta$ for $i \in [1, n]$, and $q_i = p_{i+1}$ for $i \in [1, n-1]$. The run π is *accepting* if $p_1 = q_0$ and $q_n \in F$. A word is *accepted* by $\mathcal{A}$ if there exists an accepting run on it. We denote by $\mathcal{L}(\mathcal{A})$ the set of words accepted by $\mathcal{A}$. Additionally, we use $\mathrm{ACC}_{\mathcal{A}}(w)$ to denote the set of all accepting runs of $\mathcal{A}$ on w. We omit the subscript $\mathcal{A}$ when it is clear from the context. An NFA $\mathcal{A}$ is *k-ambiguous* if, for every word $w \in \Sigma^*$, it holds that $|\mathrm{ACC}(w)| \leq k$. It is unambiguous when $k = 1$. Moreover, $\mathcal{A}$ is *finitely-ambiguous* if it is k-ambiguous for some $k \in \mathbb{Z}_+$. We use UFA and FNFA to refer to unambiguous and finitely-ambiguous NFA, respectively.

For $q \in Q$, let $\mathcal{A}_q$ denote the automaton $\mathcal{A}$ with q as its initial state, and for $S \subseteq \Delta$, let $\mathcal{A}_S$ denote the automaton obtained from $\mathcal{A}$ by restricting its transitions to S. For $\tau \in S$, we often say τ is nondeterministic in S to mean τ is nondeterministic in $\mathcal{A}_S$. Δ induces a transition function $\delta_{\mathcal{A}} : 2^Q \times \Sigma^* \mapsto 2^Q$, where $\delta_{\mathcal{A}}(Q', w)$ is the set of all states where runs on word w can end up in when starting from any state $q \in Q'$. Formally, for all $Q' \subseteq Q$, $\delta_{\mathcal{A}}(Q', \epsilon) = Q'$, and for all $w \in \Sigma^*$ and $\sigma \in \Sigma$, $\delta_{\mathcal{A}}(Q', w \cdot \sigma) = \{q \mid (p, \sigma, q) \in \Delta \text{ and } p \in \delta_{\mathcal{A}}(Q', w)\}$. When $Q' = \{p\}$, we often abuse notation and write $\delta_{\mathcal{A}}(p, w)$ instead of $\delta_{\mathcal{A}}(\{p\}, w)$, and we omit the subscript $\mathcal{A}$ when it is clear from the context. An NFA is *trim* if every state $q \in Q$ is reachable from the initial state, and can reach an accepting state, i.e., $q \in \delta(q_0, w)$ and $\delta(q, w') \cap F \neq \emptyset$ for some words w, w'.

Probabilistic Finite Automata. A *probabilistic finite automaton* (PFA) $\mathcal{P}$ is an extension of nondeterministic finite automaton that assigns an acceptance probability to each word. Specifically, $\mathcal{P}$ is an NFA augmented with an assignment $\Theta : \Delta \to [0, 1]_{\mathbb{R}}$, where for every $(p, \sigma) \in Q \times \Sigma$, $\sum_{\tau=(p,\sigma,q)\in\Delta} \Theta(\tau) = 1$. Accordingly, $\mathcal{P}$ is a 6-tuple of the form $(\Sigma, Q, q_0, \Delta, F, \Theta)$. Given a transition $\tau = (p, \sigma, q)$, the value $\Theta(\tau)$ represents the probability

of transitioning from state p to state q upon reading the symbol σ. With a slight abuse of language, we also call (p, σ, q, d) a transition of $\mathcal{P}$, where $d = \Theta(p, \sigma, q)$. We say that $\mathcal{P}$ is *based on* the NFA $\mathcal{A} = (\Sigma, Q, q_0, \Delta, F)$ and call this its *underlying* NFA.[1]

We use $\mathcal{P}(w)$ to denote the probability that $\mathcal{P}$ accepts word w where:

$$\mathcal{P}(w) := \sum_{\pi=\tau_1 \dots \tau_n \in \mathrm{ACC}(w)} \prod_{i=1}^{n} \Theta(\tau_i).$$

For every threshold $\lambda \in [0,1]_{\mathbb{R}}$, let $\mathcal{L}(\mathcal{P}_{\geq\lambda})$ denote the set of all words accepted with probability at least λ, that is: $\mathcal{L}(\mathcal{P}_{\geq\lambda}) := \{w \in \Sigma^* \mid \mathcal{P}(w) \geq \lambda\}$.

We also use $\mathcal{P}_{\geq\lambda}$ to refer to the PFA $\mathcal{P}$ with a given threshold λ. Similarly, we define $\mathcal{P}_{\leq\lambda}$, $\mathcal{P}_{>\lambda}$, and $\mathcal{P}_{<\lambda}$. A PFA $\mathcal{P}$ is called *simple* if, for every transition (p, a, q, d) of $\mathcal{P}$, the probability d belongs to $\{0, \frac{1}{2}, 1\}$. Since transitions with $d = 0$ do not affect the language accepted by a PFA, we assume $d \neq 0$ for all transitions in the remainder of this paper.

Stochastic Resolvers. Given an NFA $\mathcal{A}$, a (memoryless) *stochastic resolver* $\mathcal{R}$ for $\mathcal{A}$ resolves nondeterminism during the run on a word on-the-fly, randomly, and positionally. Formally, $\mathcal{R} : \Delta \mapsto [0,1]$, such that $\sum_{\tau=(p,\sigma,q)\in\Delta} \mathcal{R}(\tau) = 1$ holds for every pair $(p, \sigma) \in Q \times \Sigma$. A resolver $\mathcal{R}$ for $\mathcal{A}$ turns it into a PFA $\mathcal{P}^{\mathcal{A}}_{\mathcal{R}}$. We call $\{\tau \in \Delta \mid \mathcal{R}(\tau) > 0\}$ the *support* of $\mathcal{R}$.

Given a threshold $\lambda \in [0,1]$, we say that $\mathcal{A}$ is *λ-memoryless stochastically resolvable* (or simply *λ-resolvable*) if there exists a resolver $\mathcal{R}$ for $\mathcal{A}$ such that $\mathcal{L}(\mathcal{P}'_{\geq\lambda}) = \mathcal{L}(\mathcal{A})$, where $\mathcal{P}' = \mathcal{P}^{\mathcal{A}}_{\mathcal{R}}$. An automaton $\mathcal{A}$ is *positively memoryless stochastically resolvable* (or simply *positively resolvable*) if it is λ-resolvable for some $\lambda \in (0,1]$.

3 λ-Resolvability

In this section we demonstrate the importance of the threshold λ: we first show that the choice of this threshold defines a spectrum of automata classes in terms of their resolvability. We then turn to the main negative result, that checking λ-resolvability is undecidable.

▶ **Theorem 1.** *For every $\lambda \in (0,1)_{\mathbb{Q}}$ there exists a unary NFA $\mathcal{A}_\lambda$ such that $\mathcal{A}_\lambda$ is λ-resolvable but not $(\lambda+\epsilon)$-resolvable for any $\epsilon > 0$.*

Proof. Let $\lambda = \frac{m}{n}$ where $m, n \in \mathbb{Z}_+$ and $m < n$. The NFA $\mathcal{A}_\lambda$ over unary alphabet $\{a\}$ is as follows. In the initial state q_0 upon reading the letter a, $\mathcal{A}_\lambda$ nondeterministically goes to one of n branches, each with $\binom{n}{m}$ states. Consider an arbitrary ordering of all m sized subsets of $[n]$, $S_1, \dots, S_{\binom{n}{m}}$. For any branch $j \in [n]$ and $i \in [\binom{n}{m}]$, the i-th state on the j-th branch - $q^j_{S_i}$, is accepting iff $j \in S_i$. Refer to Figure 2 for an example with $\lambda = \frac{2}{3}$. The automaton $\mathcal{A}_\lambda$ is m-ambiguous, where each word in $\{a^i | i \in [\binom{n}{m}]\}$ has exactly m accepting runs. Moreover, $\mathcal{A}_\lambda$ is $\frac{m}{n}$-resolvable, since the uniform resolver that picks any transition at q_0 with probability $\frac{1}{n}$, accepts every word with probability exactly $\frac{m}{n}$. For any other resolver $\mathcal{R}$, consider the m branches $i_1, \dots, i_m$, with the lowest assigned probabilities. Let ℓ be the unique index such that $S_\ell = \{i_1, \dots, i_m\}$. Then for the word a^ℓ, we have $\mathcal{P}^{\mathcal{A}_\lambda}_{\mathcal{R}}(a^\ell) < \frac{m}{n}$. Hence, $\mathcal{A}_\lambda$ is not $(\lambda+\epsilon)$-resolvable for any $\epsilon > 0$. ◀

[1] The condition $\sum_{\tau=(p,\sigma,q)\in\Delta} \Theta(\tau) = 1$ may not hold for some pairs (p, σ) if the underlying NFA is not complete. Any NFA can be made complete by introducing a non-accepting sink state. Thus, throughout this paper, we assume that the underlying NFA of every PFA is implicitly made complete.

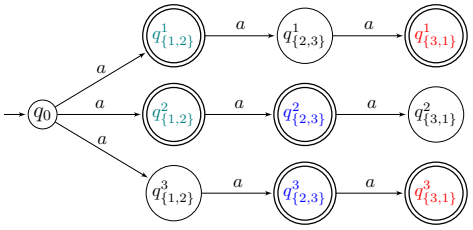


Figure 2 The automaton $\mathcal{A}_\lambda$ for $\lambda = \frac{2}{3}$, in the proof of Theorem 1.

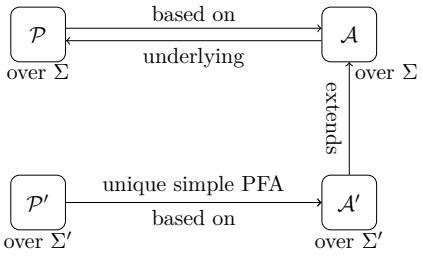


Figure 3 The construction for Theorem 2.

▶ **Theorem 2.** *The λ-resolvability problem is undecidable for NFA.*

Our approach relies on a reduction from the undecidable universality problem for simple[2] PFAs [37, 19] to the λ-resolvability problem for an NFA. Consider a simple PFA $\mathcal{P}$ over the alphabet Σ, whose underlying NFA is $\mathcal{A}$. For a simple PFA, the structure of its underlying NFA uniquely determines the probabilities on its transitions: for each state q and letter σ, there are either one or two outgoing transitions from q labelled with σ; if there is one transition it must have probability 1 and if there are two, each must have probability $\frac{1}{2}$.

Intuitively, we want the resolver (if it exists) for the underlying automaton $\mathcal{A}$ of $\mathcal{P}$ to always induce a simple PFA. So, $\mathcal{A}$ is λ-resolvable if and only if $\mathcal{P}_{\geq\lambda}$ and $\mathcal{A}$ recognise the same language. In principle, a resolver for an NFA does not necessarily enforce that two outgoing transitions both have probability $\frac{1}{2}$ and may use any arbitrary split. To enforce an equal split, we construct an NFA $\mathcal{A}'$ over an extended alphabet Σ', such that $\mathcal{A}'$ is a *super-automaton* of $\mathcal{A}$. By this, we mean that the alphabet, state set, set of accepting states, and transition set of $\mathcal{A}$ are all subsets of those of $\mathcal{A}'$, and we refer to $\mathcal{A}$ as a sub-automaton of $\mathcal{A}'$. The constructed NFA $\mathcal{A}'$ is designed so that it admits only one possible $\frac{1}{4}$-resolver, and that the resolver matches the underlying probabilities of the simple PFA $\mathcal{P}$ on the part of the automaton of $\mathcal{A}'$ corresponding to $\mathcal{A}$. These relations are described in Figure 3.

Our construction has the property that $\mathcal{A}'$ is $\frac{1}{4}$-resolvable if and only if $\mathcal{P}_{\geq\frac{1}{2}}$ is universal. Essentially, the only possible resolver induces a probabilistic automaton $\mathcal{P}'$ on $\mathcal{A}'$ such that $\mathcal{P}'(w' \cdot w) = \frac{1}{2} \cdot \mathcal{P}(w)$ for all words $w \in \mathcal{L}(A)$, where w' is a fixed word over $\Sigma' \setminus \Sigma$. This ensures $\mathcal{P}(w) \geq \frac{1}{2} \iff \mathcal{P}'(w' \cdot w) \geq \frac{1}{4}$ for $w \in \mathcal{L}(A)$. The construction itself introduces new words, however these words are $\frac{1}{4}$-resolvable by construction.

Proof Sketch of Theorem 2. Without loss of generality, we fix $\Sigma = \{a, b\}$. $\mathcal{P}_{\geq\frac{1}{2}}$ is not universal if $\mathcal{A}$ is not, and since universality for NFA is decidable, we assume that $\mathcal{L}(\mathcal{A}) = \Sigma^*$. We construct a super-automaton $\mathcal{A}'$ of $\mathcal{A}$ such that it satisfies the following three properties:

C.1 For every state p of $\mathcal{A}'$ and every $\sigma \in \Sigma'$, $\mathcal{A}'$ has at most two distinct transitions of the form (p, σ, q) for some state q.

C.2 If $\mathcal{A}'$ is $\frac{1}{4}$-resolvable, then its $\frac{1}{4}$-resolvable solution must be a simple PFA.

C.3 The language $\mathcal{L}(\mathcal{A}')$ is the disjoint union of two languages, $\mathcal{L}_{\text{ext}}$ (essentially an extension of $\mathcal{L}(\mathcal{A})$) and $\mathcal{L}_{\text{ind}}$ (essentially corresponding to the new part of $\mathcal{A}'$), satisfying:

- **(a)** For every $w \in \mathcal{L}_{\text{ind}}$, $\mathcal{P}'(w) \geq \frac{1}{4}$.
- **(b)** For every $w'' \in \mathcal{L}_{\text{ext}}$, where $\mathcal{L}_{\text{ext}} = S \cdot \Sigma^*$ for some singleton set $S = \{w'\}$ over $\Sigma' \setminus \Sigma$ and $w'' = w' \cdot w$ with $w \in \Sigma^*$, we have $\mathcal{P}'(w'') = \frac{1}{2} \cdot \mathcal{P}(w)$.

[2] Note that [19] proves the (strict) emptiness problem for simple PFA is undecidable. Since a PFA is universal if and only if its complement is empty, the universality problem is also undecidable.

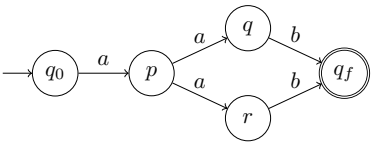


Figure 4 NFA $\mathcal{A}$.

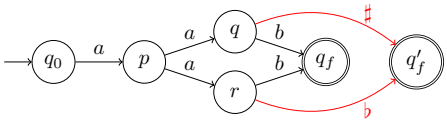


Figure 5 NFA $\mathcal{B}$.

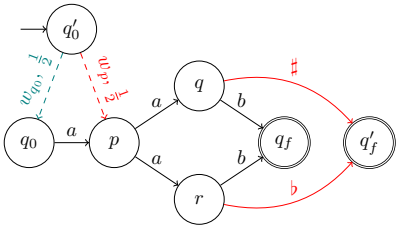


Figure 6 Partial structure of NFA $\mathcal{A}'$.

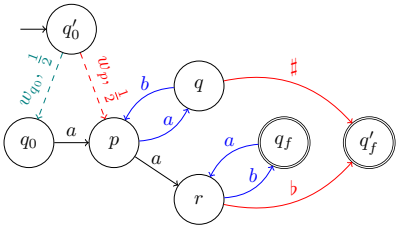


Figure 7 Additional words.

Properties C.1 and C.2 ensure that the simple PFA $\mathcal{P}'$ based on $\mathcal{A}'$ is the unique $\frac{1}{4}$-resolvable solution to $\mathcal{A}'$ if it is $\frac{1}{4}$-resolvable. Notice that regardless of whether $\mathcal{A}'$ has a $\frac{1}{4}$-resolvable solution, $\mathcal{P}'$ is based on $\mathcal{A}'$, and for each word $w \in \mathcal{L}(\mathcal{A}')$, $\mathcal{P}'$ returns its acceptance probability, which may be less than $\frac{1}{4}$. Property C.3 states that $\mathcal{P}'$ is a $\frac{1}{4}$-resolvable solution to $\mathcal{A}'$ if and only if $\mathcal{P}_{\geq \frac{1}{2}}$ is universal. Accordingly, we have:

▶ **Lemma 3.** *Let $\mathcal{A}$ be the underlying NFA of a given simple PFA $\mathcal{P}$, and let $\mathcal{A}'$ be the super-automaton of $\mathcal{A}$ satisfying properties C.1–C.3. Then $\mathcal{P}_{\geq \frac{1}{2}}$ is universal iff $\mathcal{A}'$ is $\frac{1}{4}$-resolvable.*

By Lemma 3, we can prove the undecidability of λ-resolvability by constructing, for any simple PFA $\mathcal{P}$ with underlying NFA $\mathcal{A}$, an NFA $\mathcal{A}'$ satisfying C.1–C.3. In the remainder of this section, we show how to do this and give intuitions for why these properties hold for the constructed automata. Property C.1 will hold trivially by construction, so we will focus on C.2 and C.3.

The automaton $\mathcal{A}'$ will consist of two sub-automata $\mathcal{A}_{\text{ind}}$ and $\mathcal{A}_{\text{ext}}$. The first one, $\mathcal{A}_{\text{ind}}$ recognises the language $\mathcal{L}_{\text{ind}}$, which does not depend on the structure of the original NFA $\mathcal{A}$, but rather on the numbers of $\mathcal{A}$'s states and transitions. In contrast, $\mathcal{A}_{\text{ext}}$ is an extension (super-automaton) of $\mathcal{A}$ corresponding to the language $\mathcal{L}_{\text{ext}}$.

Consider the NFA $\mathcal{A}$ given in Figure 4, where $\mathcal{L}(A) = \{aab\}$. This NFA is $\frac{1}{2}$-resolvable. Moreover, every PFA based on $\mathcal{A}$ is a $\frac{1}{2}$-resolvable solution, which is neither unique nor necessarily simple. However, if we add two transitions to $\mathcal{A}$ and derive the NFA $\mathcal{B}$ as shown in Figure 5, where for every word $w \in \{aa\sharp, aab\} \subsetneq \mathcal{L}(B)$, the accepting run on w is unique, then the $\frac{1}{2}$-resolvable solution PFA to $\mathcal{B}$ becomes both unique and simple. Note that we not only introduce new transitions but also add a newly accepting state q'_f to ensure that no additional words over $\{a, b, \sharp, \flat\}$ are introduced in a more general case. This example illustrates how property C.2 can be enforced.

More specifically, referring to Figure 6, if $\mathcal{A}$ has nondeterministic transitions to states q and r from p on reading a, we introduce two new transitions, $(q, \sharp, q'_f)$ and $(r, \flat, q'_f)$, in $\mathcal{A}'$, where $\sharp$ and $\flat$ are newly introduced symbols corresponding to the original transitions (p, a, q) and (p, a, r), respectively. If we can ensure that there exists a unique word w_p corresponding to state p, such that there is a run from the initial state q'_0 of $\mathcal{A}'$ to p with probability exactly $\frac{1}{2}$, and that the accepting runs on the words $w_p \cdot a \cdot \sharp$ and $w_p \cdot a \cdot \flat$ are unique, then we can guarantee that if the PFA $\mathcal{P}'$ is a $\frac{1}{4}$-resolvable solution to $\mathcal{A}'$, then the transition probabilities

for both (p, a, q) and (p, a, r) in $\mathcal{P}'$ must be $\frac{1}{2}$. The details of the run on the word w_p are shown in Figure 8. For words of the form $\$_0\$_i\$_j$ where $i = 1, 2$ and $j = 3, \ldots, 6$, the NFA in this figure has a unique accepting run. Analogous to the NFA $\mathcal{B}$, the symbols $\$_0, \ldots, \$_6$, p, and q_0 are newly introduced and are not in Σ. Every PFA based on this automaton is a $\frac{1}{4}$-resolvable solution if the transitions labelled with $\$_0$, $\$_1$, and $\$_2$ have probability exactly $\frac{1}{2}$. For each state p, we define $w_p = \$_0 p$, treating p as a symbol. Consequently, the run of $\mathcal{P}'$ from q'_0 to p on w_p has probability exactly $\frac{1}{2}$. We apply this technique to every nondeterministic transition in $\mathcal{A}$. Note that the automaton shown in Figure 8 is a part of $\mathcal{A}'$, is disjoint from $\mathcal{A}$, and connects to $\mathcal{A}$ through transitions of the form (y, p, p) for $p \in Q$. Hence, if $\mathcal{A}'$ has a $\frac{1}{4}$-resolvable solution $\mathcal{P}'$, then for all transitions inherited from $\mathcal{A}$, their probability values in $\mathcal{P}'$ must be $\frac{1}{2}$. Accordingly, this construction ensures that property C.2 holds.

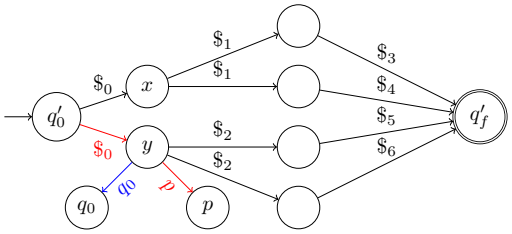


Figure 8 Details of the runs on words $w_p = \$_0 p$ and $w_{q_0} = \$_0 q_0$.

Apart from the states of $\mathcal{A}$ that have outgoing nondeterministic transitions, we also introduce a transition (y, q_0, q_0) to $\mathcal{A}'$, regardless of whether q_0 itself has nondeterministic transitions. See Figure 6 and Figure 8. Consequently, the probability of the run of $\mathcal{P}'$ from q'_0 to q_0 on the word $w_{q_0} = \$_0 q_0$ is exactly $\frac{1}{2}$. Moreover, for every word $w \in \Sigma^*$, we have $w_{q_0} \cdot w \in \mathcal{L}(\mathcal{A}')$ if and only if $w \in \mathcal{L}(A)$. Hence, property C.3b follows.

It is important to explain why we consider the $\frac{1}{4}$-resolvable problem of $\mathcal{A}'$ instead of the $\frac{1}{2}$-resolvable one, as in the previous example with the NFA $\mathcal{B}$. Refer to Figure 7 for a modified example. Our intention is to force the $\frac{1}{2}$ probability split using words of the form $w_p \cdot w \cdot \sigma'$, where $|w| = 1$ and $\sigma' \in \{\sharp, \flat\}$, however the construction introduces other words that need to be managed. For example, the words $w_p \cdot a \cdot (ab)^i \cdot \sharp$ and $w_p \cdot a \cdot (ab)^i \cdot \flat$ belong to $\mathcal{L}(\mathcal{A}')$ for all $i \in \mathbb{Z}_+$. However, for the simple PFA $\mathcal{P}'$ based on $\mathcal{A}'$, we have $\mathcal{P}'(w_p \cdot a \cdot (ab)^i \cdot \sharp) = \frac{1}{2^{i+1}} < \frac{1}{2}$ and $\mathcal{P}'(w_p \cdot a \cdot (ab)^i \cdot \flat) \geq \frac{1}{2}$ for all $i \in \mathbb{Z}_+$. Even if we set the probability of the run of $\mathcal{P}'$ from the state q'_0 to p on w_p to be 1, we still have $\mathcal{P}'(w_p \cdot a \cdot (ab)^i \cdot \sharp) \to 0$ as $i \to \infty$. That is, when we introduce the transitions $(q, \sharp, q'_f)$ and $(r, \flat, q'_f)$, we may also introduce additional words into $\mathcal{L}(\mathcal{A}')$. However, based on our tentative construction of $\mathcal{A}'$, it is possible that $\mathcal{L}(\mathcal{P}'_{\geq \frac{1}{2}}) \subsetneq \mathcal{L}(\mathcal{A}')$, regardless of whether $\mathcal{P}$ is universal or not. The words in $\mathcal{L}(\mathcal{A}') \setminus \mathcal{L}(\mathcal{P}'_{\geq \frac{1}{2}})$ are of the form $\$_0 \cdot p \cdot w \cdot \sigma'$, where $p \in Q$, $w \in \Sigma^*$ with $|w| \neq 1$, and σ' is a symbol introduced for a nondeterministic transition, such as $\sharp$ or $\flat$, etc.

To address this issue, we ensure the probability is at least $\frac{1}{4}$ probability when $|w| \neq 1$, we construct a DFA $\mathcal{A}_{\text{extra}}$ recognising the language $\{\$_0\} \cdot Q \cdot (\Sigma^* \setminus \Sigma) \cdot \{\sharp, \flat, \ldots\}$ and add it to $\mathcal{A}'$. The automaton $\mathcal{A}_{\text{extra}}$ includes a unique transition from the initial state q'_0 to a state x upon reading symbol $\$_0$. This transition $(q'_0, \$_0, x)$ is the same as the one shown in Figure 8, with the rest of $\mathcal{A}_{\text{extra}}$ continuing from x. For every simple PFA based on the automaton given in Figure 8, the probability value of the transition $(q'_0, \$_0, x)$ is $\frac{1}{2}$. Thus, for every additional word $w \notin \mathcal{L}_{\text{ext}}$, we have $\mathcal{P}'(w) \geq \frac{1}{4}$, ensuring that property C.3a holds. Hence, the constructed $\mathcal{A}'$ satisfies properties C.1–C.3.

From Lemma 3, it follows that the λ-resolvability problem for NFA is undecidable. ◀

4 Deciding Resolvability for Unary Finite Automata

▶ **Theorem 4.** *Positive resolvability is in* $\mathsf{NP}^{\mathsf{coNP}}$ *for unary NFA.*

For a unary NFA a memoryless stochastic resolver induces a unary probabilistic automaton, or equivalently a Markov chain. This is given by an initial distribution as a row vector $I \in [0,1]^d$, a stochastic matrix $P \in [0,1]^{d\times d}$ and a final distribution as a column vector $F \in [0,1]^d$ such that $\mathcal{P}(a^n) = IP^nF$. We make use of the following lemma, which summarises standard results in Markov chain theory to our needs (see e.g. [30, Sec 1.2-1.5] or [34, Sec 1.8]). The lemma shows that there are a sequence of limit distributions that are visited periodically. Essentially, we get, for each word length n, a set of reachable states, and except for a possible initial sequence, these sets of states repeat periodically. Once in the periodical part, whenever there is a reachable accepting state, there must be some reachable accepting state in a bottom strongly connected component (SCC) to ensure the probability is bounded away from zero. The key point of the following lemma is to translate this into the support of the limit distributions for these states to be non-zero / zero for reachable states in- / outside of bottom SCCs, so that we only need to consider the support graph of the Markov chain, not the probabilities themselves. This allows us to guess the support of the resolver in NP and verify in coNP that any resolver with this support would suffice.

▶ **Lemma 5.** *Let P be a d-dimensional Markov chain with initial distribution I. There exists T, bounded by an exponential in d, and limit distributions $\pi^{(0)}, \dots, \pi^{(T-1)} \in [0,1]^d$, such that $\lim_{n\to\infty} IP^k(P^T)^n = \pi^{(k)}$. Furthermore, $\pi^{(k)}(q)$ is non-zero if and only if*

1. *q is in a bottom SCC, and*
2. *q is reachable in P from some state in I by some path of length $k + T\ell$ for some $\ell \le d$.*

We now show how to decide whether there is a stochastic memoryless resolver.

Proof of Theorem 4. Let $\mathcal{A} = (\{a\}, Q, q_0, \Delta, F)$ be a unary NFA over the alphabet $\{a\}$, with $|Q| = d$, initial state q_0 encoded in a row vector $I \in \{0,1\}^d$ and final states encoded in a column vector $F \in \{0,1\}^d$. We ask if there is a $P \in [0,1]^{d\times d}$ with support at most Δ and λ such that $IP^nF \ge \lambda$ whenever $a^n \in \mathcal{L}(\mathcal{A})$.

If P exists, it has some support $S \subseteq \Delta$ and we guess this support of P non-deterministically. The choice of support induces a new NFA $\mathcal{A}_S \subseteq \mathcal{A}$, in which every edge will be attributed a non-zero probability and $\mathcal{L}(\mathcal{A}_S) = \{a^n \mid IP^nF > 0\}$. We verify, in coNP [43], that $\mathcal{L}(\mathcal{A}_S) = \mathcal{L}(\mathcal{A})$. If not, this is not a good support.

Henceforth, we assume $\mathcal{L}(\mathcal{A}_S) = \mathcal{L}(\mathcal{A})$ and consider the condition to check whether the support S is good. For any P with support S we have $IP^nF > 0$ for $a^n \in \mathcal{L}(\mathcal{A}_S)$. It remains to decide whether this probability can be bounded away from zero, that is for some $\lambda > 0$, $IP^nF > \lambda$ whenever $a^n \in \mathcal{L}(\mathcal{A}_S)$. The condition we describe is independent of the exact choice of P and is based only on the structure of $\mathcal{A}_S$. Using T from Lemma 5 we decompose $\mathcal{L}(\mathcal{A}_S)$ into T phases and define for all $k \in \{0, \dots, T-1\}$ the set $\mathcal{L}_k = \{a^{k+Tn} \mid n \in \mathbb{N}\} \cap \mathcal{L}(\mathcal{A})$. Each $\mathcal{L}_k$ is either finite, in which case it turns out there is nothing to do, or eventually periodic with period T, in which case we use Lemma 5's characterisation of non-zero limit:

$\mathcal{L}_k$ is finite. There is nothing to decide: any P with support S induces a lower bound $\min\{IP^k(P^T)^nF \mid n \in \mathbb{N}, a^{k+Tn} \in \mathcal{L}_k\} > 0$ on the weight of the words in $\mathcal{L}_k$, as the minimum over a finite set of non-zero values.

$\mathcal{L}_k$ is eventually periodic. We check there exists a path from q_0 to some final state $q_f \in F$ of length $k + T\ell$ for some $\ell \le d$, and that q_f is in some bottom SCC in $\mathcal{A}_S$. Then for any P with support S, Lemma 5 entails a limit distribution $\pi^{(k)} \in [0,1]^d$ such that $\lim_{n\to\infty} IP^k(P^T)^n = \pi^{(k)}$, in which $\pi^{(k)}(q_f)$ is non-zero. Thus $\lim_{n\to\infty} IP^k(P^T)^nF =$

$\pi^{(k)}F = \epsilon_k > 0$. In particular, there exists n_k such that, for all $n' > n_k$, we have $IP^k(P^T)^{n'}F > \epsilon_k/2$. Hence, there is a lower bound on the weights of words in $\mathcal{L}_k$: $\min\{IP^k(P^T)^nF \mid n : a^{k+Tn} \in \mathcal{L}_k, n \leq n_k\} \cup \{\epsilon_k/2\} > 0$, which is a minimum over a finite set of non-zero values.

If there is no such path, then for any P with support S, the limit distribution is zero in all final states: the probability tends to zero no matter the exact choice of probabilities. For a given k, it can be decided in polynomial time whether $\mathcal{L}_k$ is finite and whether there is a path of the given type. On a boolean matrix A_S representing the reachability graph of $\mathcal{A}_S$, the vector $I(A_S)^k$ and boolean matrix $(A_S)^T$ can be computed by iterated squaring in which the reachability questions can be checked.

Thus to summarise, our procedure guesses a support $S \subseteq \Delta$ and the period T in NP, and verifies in coNP both that $\mathcal{L}(\mathcal{A}_S) = \mathcal{L}(\mathcal{A})$ and guess the $k < T$ to verify the required reachability queries for k. ◀

In Theorem 11 we show that the problem is coNP-hard already in the finite-ambiguous case, we present this result alongside the non-unary case in Section 6 due to commonalities.

5 Unambiguous Finite Automata

Given an NFA $\mathcal{A}$, a support S is *bad* if there is no resolver $\mathcal{R}$ over S that positively resolves $\mathcal{A}$, that is, $\mathcal{P}^{\mathcal{A}}_{\mathcal{R}}$ is not a positively resolvable solution to $\mathcal{A}$ for each $\mathcal{R}$ over S. We start this section with a property related to bad supports for finitely-ambiguous automata, which we use to derive the decidability and complexity bounds for the resolvability problems.

▶ **Lemma 6.** *Let S be a support for an FNFA $\mathcal{A}$ with $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}_S)$. For a word w with k accepting runs, $\pi_1, \ldots, \pi_k$, let $b(w)$ be the minimum number of nondeterministic transitions present in any of the π_i. Then S is a bad support if and only if there is an infinite sequence $\{w_i\}_i$ of words such that $\{b(w_i)\}_i$ is a strictly increasing sequence.*

For a UFA– where each word has exactly one accepting run – removing a transition without changing the language implies that the transition either originates from an unreachable state or leads to a state with empty language. Such transitions can thus be safely removed or assigned probability zero. Assuming the UFA is trim, its support is unique. It is not positively resolvable if and only if this support is bad.

It is easy to see that a UFA is not positively resolvable if it admits an accepting run of the form $q_0 \xrightarrow{x} q \xrightarrow{y} q \xrightarrow{z} f$, where q_0 is the initial state, f is an accepting state, and there is a nondeterministic transition taken while reading the infix y. By pumping y, we obtain a sequence of accepted words of the form xy^iz, each with a unique accepting run, but with decreasing probabilities assigned by any stochastic resolver. This follows from the fact that the loop induced by y contains a nondeterministic transition, which causes the probability mass to diminish. The UFA in Figure 1b illustrates this phenomenon: it is unambiguous but not positively resolvable. This is witnessed by the word bb, where the first b corresponds to the nondeterministic, pumpable part of the word.

In the following, we show that the existence of such a witness word is both a necessary and sufficient condition for a UFA not to be positively resolvable.

▶ **Lemma 7.** *A UFA $\mathcal{A}$ is not positively resolvable if and only if there exists a word $w = xyz \in \mathcal{L}(\mathcal{A})$ such that the accepting run on w satisfies the following conditions:*

1. *After reading the prefix x, the run reaches a state $q \in \delta_{\mathcal{A}}(q_0, x)$, and after subsequently reading y, it returns to the same state, i.e., $q \in \delta_{\mathcal{A}}(q, y)$. Moreover, reading z from q leads to an accepting state: $\delta_{\mathcal{A}}(q, z) \cap F \neq \emptyset$.*
2. *The words y and z begin with the same letter a, it follows that the transition taken when reading y from q is nondeterministic.*

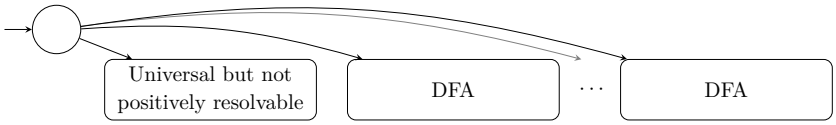


Figure 9 Conceptual idea of hardness in both unary and non-unary cases (exact details differ).

By guessing this witnessing word of Lemma 7 letter by letter, we have an NL procedure for deciding whether a UFA is positively resolvable or not.

▶ **Theorem 8.** *The positive resolvability problem for UFA is in NL.*

We can show a matching lower bound in the unambiguous case by a straightforward reduction from the emptiness problem for UFA, which is known to be NL-complete.

▶ **Theorem 9.** *Given a UFA, it is NL-hard to decide if it is positively resolvable, and NL-hard to decide if it is λ-resolvable for any fixed $\lambda > 0$.*

For a positively resolvable UFA $\mathcal{A}$, Lemma 7 essentially states that no accepting run may contain nondeterministic transitions in its looping part. This is equivalent to requiring that all transitions within SCCs of $\mathcal{A}$ are deterministic. Exploiting this structural property, we can compute the largest possible value of λ such that $\mathcal{A}$ is λ-resolvable. To do so, we collapse all SCCs into single nodes, yielding a rooted directed acyclic graph (DAG), and reduce the problem to counting the maximum number of disjoint paths in this DAG, which can be computed in polynomial time.

▶ **Theorem 10.** *It is polynomial time to decide whether a UFA is λ-resolvable.*

6 Finitely-Ambiguous Finite Automata

In this section we establish decidability and hardness for both positive resolvability and λ-resolvability for finitely-ambiguous NFA.

6.1 Hardness of Resolvability

We show that when an NFA is finitely-ambiguous, deciding the existence of a resolver, and the existence of a λ-resolver are both coNP-hard in the unary case and PSPACE-hard otherwise. While the unary and non-unary cases differ slightly, both are based on deciding whether the union of k given DFAs are universal.

The ideas of both proofs are the same, we will extend the given automaton with a new component that ensures it is universal, but the new component itself will not be positively resolvable (see Figure 9). Therefore, if the existing k-DFAs are universal then the resolver that uniformly resolves between these k-DFAs (assigning zero weight to the new component) is sufficient. However, if the k-DFAs are not themselves universal, then the new component must be used to recognise some words, but this component is not positively resolvable, and so neither is the whole automaton. The difference between the proofs will be how we ensure this works even if only a single short word is not in the union of the k-DFAs.

In the unary case we reduce from the problem of whether a unary NFA is universal which is coNP-hard [43], but similarly to [13, Theorem 7.20], we can assume that the shape is the union of simple cycles. By bringing into Chrobak normal form in polynomial time the automaton has an initial stem and a nondeterministic transition to k cycles at the end

of the stem. But since universality requires that all words are accepting, each state of the stem must be accepting, which can be pre-checked in polynomial time. Thus the hard case remains to determine whether the union of the remaining k-cycle DFAs are universal.

▶ **Theorem 11.** *Given a unary k-ambiguous NFA it is* coNP*-hard to decide if it is positively resolvable, and* coNP*-hard to decide if it is $1/(k-1)$-resolvable for $1 < k \in \mathbb{Z}$.*

In the non-unary case we show PSPACE-hardness via the problem asking whether the union language of k-DFAs is universal, which is PSPACE-complete. This problem can be seen as the complement of the (well-known) problem of intersection emptiness for DFAs, that is, whether the intersection of k (not fixed) DFAs is empty, which is known to be PSPACE-complete [26]. The unary case exploits the cyclic structure to ensure that if some word is not in the union, then there are indeed infinitely many missing words. There is no such natural analogue in the non-unary case requiring a modified construction.

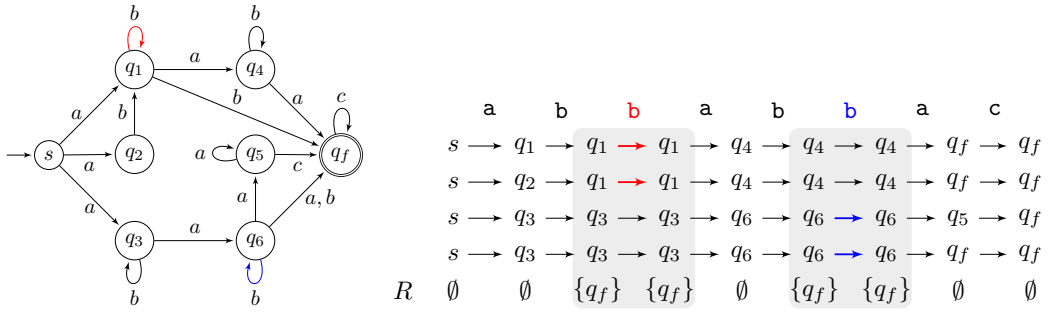
▶ **Theorem 12.** *Given a k-ambiguous NFA, it is* PSPACE*-hard to decide if it is positively resolvable, and* PSPACE*-hard to decide if it is $1/(k-2)$-resolvable for $2 < k \in \mathbb{Z}$.*

6.2 Deciding Positive Resolvability for Finitely-Ambiguous NFA

In Section 5, we established positive resolvability for unambiguous automata by identifying a witnessing "bad" word. In this section, we generalise the notion of a "bad" word to finitely-ambiguous NFA. For any UFA $\mathcal{A}$, there exists a unique support S such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}_S)$ assuming $\mathcal{A}$ is trim. However, when $\mathcal{A}$ is no longer unambiguous, there may be multiple such supports S satisfying $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}_S)$. If $\mathcal{A}$ is not positively resolvable, then all of its supports are *bad*, in the sense that no stochastic resolver over any of these supports can positively resolve $\mathcal{A}$. Our algorithm for deciding positive resolvability works by checking if supports of $\mathcal{A}$ are bad. We characterise a bad support S, using a witness *bad* word that satisfies several conditions. This is a generalisation of the witnessing word in the UFA case. Intuitively, a bad word for a support is a word, which contain subwords, that can be pumped arbitrarily to produce new words with acceptance probability arbitrarily close to 0. These "pumpable" subwords must preserve some reachability behaviour as well as contain probabilistic choices with probability in $(0, 1)$.

▶ **Definition 13** (Bad word)**.** *Let $\mathcal{A}$ be an FNFA and $S \subseteq \Delta$ be a support. Then a word $w \in \Sigma^*$ with M accepting runs is a bad word for S if for some $\ell \leq M$, there exists a decomposition $w = x_0y_1x_1 \dots y_\ell x_\ell$ with words $x_0, \dots, x_\ell \in \Sigma^*$, $y_1, \dots, y_\ell \in \Sigma^+$, sets of states $R_1, \dots, R_\ell$, $Q_1, \dots, Q_\ell \subseteq Q$, and states $q_1, \dots, q_\ell$ with $q_i \in Q_i$ for each $i \in [\ell]$ such that:*

1. *For each $i \in [\ell]$, all accepting runs of w in $\mathcal{A}_S$ from q_0 end in Q_i after reading $x_0y_1 \dots x_{i-1}$, i.e., $q \in \delta_{\mathcal{A}_S}(q_0, x_0y_1 \dots x_{i-1})$ and $\delta_{\mathcal{A}_S}(q, y_i \dots x_\ell) \cap F \neq \emptyset \iff q \in Q_i$.*
2. *An accepting run of w in $\mathcal{A}_S$ must be in some q_i for some $i \in [\ell]$ after reading $x_0y_1x_1 \dots y_i$.*
3. *For each $i \in [\ell]$ and each $q \in Q_i$, every accepting run of w that reaches q after reading the prefix $x_0y_1 \dots x_{i-1}$ also returns to q after subsequently reading y_i. Moreover, when $q = q_i$, this run of y_i from q_i to q_i contains at least one transition from S which is nondeterministic in $\mathcal{A}_S$.*
4. *For each $i \in [\ell]$, R_i contains all states that are reached from q_0 after reading $x_0y_1 \dots x_{i-1}$, but has no continuation to an accepting run in $\mathcal{A}_S$ for the remaining suffix $y_ix_i \dots x_\ell$, i,e., $q \in \delta_{\mathcal{A}_S}(q_0, x_0y_1 \dots x_{i-1})$ and $\delta_{\mathcal{A}_S}(q, y_i \dots x_\ell) \cap F = \emptyset \iff q \in R_i$. Moreover, R_i is also exactly the set of states that are reached from q_0 after reading $x_0y_1 \dots x_{i-1}y_i$, with no continuation to an accepting run in $\mathcal{A}_S$ for the remaining suffix $x_i \dots x_\ell$, i.e., $q \in \delta_{\mathcal{A}_S}(q_0, x_0y_1 \dots x_{i-1}y_i)$ and $\delta_{\mathcal{A}_S}(q, x_i \dots x_\ell) \cap F = \emptyset \iff q \in R_i$.*



(a) Non-positively resolvable FNFA. **(b)** Four accepting runs on w.

Figure 10 A bad word $w = x_0y_1x_1y_2x_2 = \texttt{abbabbac}$, where $x_0 = \texttt{ab}$, $y_1 = \texttt{b}$, $x_1 = \texttt{ab}$, $y_2 = \texttt{b}$, and $x_2 = \texttt{ac}$. The set R includes all reachable states from which no accepting run exists for the remaining suffix.

▶ **Example 14.** The NFA $\mathcal{A}$ in Figure 10a is 4-ambiguous. Consider the full support S containing all transitions of $\mathcal{A}$. The word $w = \texttt{abbabbac}$ serves as a bad word witness for S, with decomposition $w = x_0y_1x_1y_2x_2$ where $\ell = 2$, $x_0 = \texttt{ab}$, $y_1 = \texttt{b}$, $x_1 = \texttt{ab}$, $y_2 = \texttt{b}$, and $x_2 = \texttt{ac}$. There are four distinct accepting runs on w; see Figure 10b for an illustration. Based on Definition 13, for w, we have $Q_1 = \{q_1, q_3\}$, $Q_2 = \{q_4, q_6\}$ and $R_1 = R_2 = \{q_f\}$. Among the four accepting runs, two take nondeterministic transitions from S when reading y_1, and the other two do so when reading y_2. Runs from Q_i to Q_i are highlighted and nondeterministic transitions are coloured in the figure. This shows that w is a bad word for S: the entire sequence of words $\{x_0y_1^ix_1y_2^ix_2\}_{i\geq 1}$ admits four accepting runs each, and the probability assigned to each such word diminishes as i increases under any stochastic resolver over S. In fact, this automaton is not positively resolvable. Although there exists another support which still preserves the language of $\mathcal{A}$ – the one that excludes the transitions going through q_2, we can show that this support is also bad, witnessed by the same bad word. With this support, we no longer have the second accepting run in Figure 10b. ⌟

Detecting bad words involves finding a decomposition as well as suitable Q_i's, q_i's and R_i's. In order to explicitly track these objects for some word, we consider an automata $\Gamma_{\mathcal{A}}$, called the *run automaton* of $\mathcal{A}$. Intuitively, a run of a word on $\Gamma_{\mathcal{A}}$, records all the states reached in accepting runs of w as well as the reachable states from which there are no continuation to accepting runs.

Run automata. Let $\mathcal{A}$ be k-ambiguous. Then the run automaton $\Gamma_{\mathcal{A}}$ has state space $Q^k \times 2^Q$, and transitions $((q_1, \ldots, q_k), R) \xrightarrow{a} ((q'_1, \ldots, q'_k), R')$ iff $q_i \xrightarrow{a} q'_i$ in $\mathcal{A}$ and $\forall q \in R, \delta(q, a) \subseteq R'$. Also, for every state $(\mathbf{q}, R)$ in $\Gamma_{\mathcal{A}}$, the set of states in the tuple $\mathbf{q}$ and the set R are disjoint. Every word $w = a_1 \ldots a_t \in \mathcal{L}(\mathcal{A})$ with k accepting runs has a run on $\Gamma_{\mathcal{A}}$ from initial state $((q_0)^k, \emptyset)$ to $(\mathbf{q}, Q')$ where $\mathbf{q} \in F^k$ and $Q' \cap F = \emptyset$ such that 1) in the ith step of the run, it reaches $((q_1^i, \ldots, q_k^i), R_i)$ if the jth accepting run of w is in q_j^i after reading $a_1 \ldots a_i$ and 2) $q \in R_i$ iff there is no accepting run of $a_{i+1} \ldots a_t$ from the reachable state q. We call such a run, a *nice run* of w in $\Gamma_{\mathcal{A}}$. For a word in $\mathcal{L}(\mathcal{A})$ with strictly fewer than k accepting runs will have such a run on $\Gamma_{\mathcal{A}}$, where for some $i_1 \neq i_2$, $q_j^{i_1} = q_j^{i_2}$ for all j, i.e. there will be at least two copies of same run. Every word w has a unique nice run on $\Gamma_{\mathcal{A}}$ up to duplication and shuffling of accepting runs. A bad word for S is essentially a word in $\mathcal{L}(A_S)$ which has a nice run on $\Gamma_{\mathcal{A}_S}$ such that in this nice run, the component of the run on the

subword y_i is a self-loop in $\Gamma_{\mathcal{A}_S}$ with several accepting runs containing a nondeterministic transition from S. In Example 14 these self loops happen at states $((q_1, q_1, q_3, q_3), \{q_f\})$ and $((q_4, q_4, q_6, q_6), \{q_f\})$ in the nice run of the bad word in the run automata.

The following lemma ensures that for the loop words y_i in Definition 13, there is always a unique loop from any state $q \in Q_i$ back to itself. Assume there are two loops from q to q over the word y. Let x be a word that there is a run from q_0 to q and z a word that there is run from q to a final state. Then, the sequence of words $w_i = xy^iz$ admits an unbounded number of accepting runs, contradicting that the automaton is finitely-ambiguous.

▶ **Lemma 15.** *In an FNFA, for any accepting run containing a loop from q to q over the word y, it is the unique loop from q to q over y.*

A nice run for a bad word contains a cycle corresponding to each subword y_i. The following lemma ensures that we can construct new words by pumping these subwords while preserving the number of accepting runs.

▶ **Lemma 16.** *Let $w = xyz$ be a word in $\mathcal{L}(\mathcal{A}_S)$ with M accepting runs, whose nice run in $\Gamma_{\mathcal{A}_S}$ has a cycle on the subword y. Then for each $j \in \mathbb{N}$, xy^jz also has M accepting runs.*

▶ **Corollary 17.** *Let $w = x_0y_1x_1 \dots x_\ell$ be a bad word for support S with M accepting runs. Then for each $(j_1, \dots, j_\ell) \in \mathbb{N}^\ell$, $x_0y_1^{j_1}x_1y_2^{j_2} \dots y_\ell^{j_\ell}x_\ell$ also has M accepting runs.*

For FNFA, a support S is bad when either $\mathcal{L}(\mathcal{A}_S) \subsetneq \mathcal{L}(\mathcal{A})$ or there is a bad word to witness that the support is bad. The following is a key lemma of this section, demonstrating that the existence of such a word is necessary and sufficient for a support to be bad.

▶ **Lemma 18** (Bad support-bad word)**.** *Let $\mathcal{A}$ be an FNFA, and let $S \subseteq \Delta$ be a support. Then, S is a bad support if and only if $\mathcal{L}(\mathcal{A}_S) \subsetneq \mathcal{L}(\mathcal{A})$ or there exists a bad word for S.*

Proof Sketch. One direction is straightforward: If there exists a bad word as defined in Definition 13, we construct a sequence of words by simultaneously pumping all the loop words. By Corollary 17, each word in this sequence has the same number of accepting runs. However, the number of nondeterministic transitions in the accepting runs of each word increases. By Lemma 6, this implies that the support is bad.

Towards the other direction, suppose we are given a bad support. Then, by Lemma 6, there exists a sequence of words whose accepting runs contain an increasing number of nondeterministic transitions. We select a word w from this sequence such that every accepting run of w contains more nondeterministic transitions than the number of states in the run automaton. This ensures that, in any nice run over w in the run automaton, each accepting run must traverse a self-loop that includes a nondeterministic transition. Formally, for the ith accepting run, the word w can be decomposed as $x_iy_iz_i$, where a nice run reaches the state $(\mathbf{q}_i, R_i)$ after reading x_i, and returns to $(\mathbf{q}_i, R_i)$ after reading the loop word y_i. Moreover, the subrun over y_i contains at least one nondeterministic transition. Since the loop segments y_i may overlap across different accepting runs, we construct a new word w' by pumping these loop segments to ensure they are disjoint. The resulting word w' satisfies all the conditions of Definition 13, and is therefore a bad word. ◀

To decide the positive resolvability of an FNFA $\mathcal{A}$, we consider all possible supports S such that $\mathcal{L}(\mathcal{A}_S) = \mathcal{L}(\mathcal{A})$, and check whether each support is bad by analysing the run automaton $\Gamma_{\mathcal{A}_S}$. According to Lemma 18, if a support is bad, there must exist a bad word for it, and there must also be a corresponding nice run over this word in $\Gamma_{\mathcal{A}_S}$. This nice run encodes the structure of all accepting runs of $\mathcal{A}_S$ on the bad word. In particular, for each accepting run π of $\mathcal{A}_S$, the nice run in $\Gamma_{\mathcal{A}_S}$ includes a loop that contains at least one

nondeterministic transition from S in its component corresponding to π. To show that a support is bad, it suffices to search for such nice runs in $\Gamma_{\mathcal{A}_S}$, which gives us a decidable procedure. A naive algorithm to check for a bad word would be to nondeterministically guess a nice run of a bad word in $\Gamma_{\mathcal{A}_S}$ from $((q_0)^k, \emptyset)$ to $(\mathbf{q}, Q')$ with $\mathbf{q} \in F^k$ and $Q' \cap F = \emptyset$. This involves guessing a letter at each step, keeping track of states reached in each run, i.e. at jth step computing the states $(q_1^j, \ldots, q_k^j)$ and guessing R_j for state $((q_1^j, \ldots, q_k^j), R_j)$ in $\Gamma_{\mathcal{A}_S}$. For the bad word decomposition, we would also need to guess the states Q_i and the state $q_i \in Q_i$ for simulating the cycle in $\Gamma_{\mathcal{A}_S}$ with nondeterministic transition in some run. We also track runs that have already seen nondeterministic transition in some cycle on $\mathcal{A}_S$. Finally, we have a bad word, if all runs have seen nondeterministic transitions in some cycle and the sets R_j's are consistent with the final guessed word.

However, given an FNFA, since its degree of ambiguity k is known to be bounded by $5^{\frac{|Q|}{2}}|Q|^{|Q|}$ [46], the size of the run automaton $\Gamma_{\mathcal{A}_S}$ can be $|Q|^k 2^{|Q|}$ in the worst case, which is doubly exponential in the size of $\mathcal{A}$. Hence the described procedure is not in PSPACE. By slightly modifying this procedure, instead of storing the state tuple $(q_1^j, \ldots, q_k^j)$ reached at every step, we store the set of states $A_j = \{q_1^j, \ldots, q_k^j\}$ along with keeping track of those states G_j in A_j, to which all accepting runs are yet to see nondeterministic transitions in some cycle in $\Gamma_{\mathcal{A}_S}$. Additionally, for the cycle part, when we guess Q_i, since from Lemma 15 we know that every state $q \in Q_i$ has unique run to itself for the subword y_i, there are at most n runs to track in this part. So we arbitrarily order the runs and track the progression of these runs in the cycle. Note that, in this part any transition doesn't change the size of the sets A_i and hence are bijections on A_i. We guess these bijections until the compositions of all the bijections is the identity map, i.e. each run has returned to the point of entering the cycle. At the end of the cycle we remove all states from G_i, whose cycles have encountered nondeterministic transitions from S. Since this procedure stores information only about sets A_j, G_j, R_j of size at most n, as well as bijections on $[n]$, the state space can be described using $O\big(n^{2n+4}\big)$, giving us a PSPACE procedure for detecting bad words.

▶ **Theorem 19.** *It can be decided in* PSPACE *if an FNFA is positively resolvable. Moreover, the shortest bad word is of length at most* $O\big(n^{2n+4}\big)$.

Proof. We provide a PSPACE algorithm for the complement problem, i.e. checking if the given FNFA is not positively resolvable. The first step is to go through all possible supports $S \subseteq \Delta$ that are language equivalent to $\mathcal{A}$. This can be checked in PSPACE[43] for general NFA.

Given a support S with $\mathcal{L}(\mathcal{A}_S) = \mathcal{L}(\mathcal{A})$, we check whether it is bad by nondeterministically guessing a bad word w. Specifically, we guess a word of the form $w = x_0 y_1 x_1 \ldots y_\ell x_\ell$, satisfying the conditions of Definition 13, letter by letter. During this process, we also guess the start and end points of each y_i, and simultaneously track an abstraction of the accepting runs, as well as of all non-accepting runs.

While reading a part x_i, which we call *branching*, or y_i, which we call *looping*, the abstraction intuitively preserves the following for a bad word: (1) which states are reachable on the overall word read so far, but only appear on rejecting runs in a set R, (2) which states are part of an accepting run of the overall word in a set A, and (3) which states are part of the accepting runs but have not yet been shown to be "bad", in a set G; formally, these are the states reached by an accepting run whose certificate state is q_j (2nd item in Definition 13) with $j > i$. The formal rules for these sets are: $R \cap A = \emptyset$ and $G \subseteq A$.

The update rules from sets R, A, G to R', A', G' when guessing a letter a are that all a-successors of a state in R must be in R', all a-successors of a state in A must be in $A' \uplus R'$ and at least one of them must be in A', and all states in G' must be a-successors of states in G.

When entering a looping part y_i, we remember the set R and denote as R_0. On reading y_i the abstraction additionally tracks (4) the set A of states that occurs on some accepting run at the beginning of y_i and remember as A_0(A_0 is the Q_i from Lemma 18), (5) which state $q' \in A_0$ is on the path for the runs with index i (that start and end with q_i in terms of Lemma 18), (6) whether this path through y_i has seen a non-deterministic transition using a boolean flag, and (7) a bijection $f\colon A_0 \to A$.

The update rules when guessing a letter a in the looping part additionally require that, for all states $q \in A$ resp. $q \in G$, exactly one a-successor is in A' resp. G'; q' is left unchanged, and the boolean flag is set to true if the a-transition from $f(q')$ is non-deterministic; otherwise it is left unchanged. The bijection f is updated to f' such that, if $f\colon q \mapsto p$, then $f'\colon q \mapsto p'$ such that $p' \in A'$ and p' is an a-successor of p.

Moving from the branching mode to looping mode can be done through an ε-transition, setting A_0 to A, f to the identity and R_0 to R, guessing $q_i \in G$ and setting the boolean flag to false. Moving back to branching mode happens through an ε-transition that can be taken when the boolean flag is set to true, f is the identity, A is A_0 and R is R_0. q' is removed from G in this ε-transition.

Finally, we accept the run when we reach a state with $R \cap F = \emptyset$ and $G = \emptyset \neq A \subseteq F$ in the branching mode.

The algorithm essentially reduces to a reachability problem on a graph with an exponential number of nodes. However, rather than explicitly constructing this graph, we operate on abstractions involving only a constant number of sets, variables, and functions, thereby requiring only polynomial space. The algorithm is then in NPSPACE, hence, in PSPACE, since NPSPACE=PSPACE[39].

To show the correctness of the algorithm, we prove that a bad word exists if and only if we can construct an accepting run in the graph. For a bad word $w = x_0y_1x_1 \dots y_\ell x_\ell$ for some ℓ with M accepting runs, we observe that the M accepting runs do not have a natural order among them, and that a repetition of each y_i may diminish the probability mass of several accepting runs. We give each accepting run an index i that refers to the first y_i that diminishes it. In particular, the accepting runs with index i are in q'_i after reading $x_0y_1 \dots y_i$. We simply take the terms from Definition 13 and construct an accepting run in the graph.

We can likewise construct a bad word from an accepting run on the graph as follows: starting with x_0, the words between two ϵ-transitions define words $x_0, y_1, x_1, y_2, \dots$; for the y_i we also store their respective A_0 and q' as Q'_i and q'_i, respectively. It is now easy to see that this satisfies all local conditions of Definition 13, so that all we need to show is that all accepting runs are in q'_i after reading $x_0y_1 \dots y_i$ for some i. But this is guaranteed by $G = \emptyset$.

We get as a side result, the shortest bad word cannot be longer than the size of the graph described, and thus, $O\big(n^{2n+4}\big)$. This is a rough bound: each state can appear in both the domain and range of the bijection, leading to at most $2n$ possibilities. There are further 2 possibilities: if a state is in the range of the bijection, then it belongs to A, and consequently, it can also be in G; if it is not in the range of the bijection, then it does not belong to A and can either be in R (since R and A are disjoint) or not. In addition, it can be the state q' that we need to track in step 5. It may be in the set R_0 or not. ◀

6.3 Deciding λ-Resolvability for Finitely-Ambiguous NFA

Here we will provide an algorithm to decide λ-resolvability for FNFA. We do this by building a finite system of inequalities and check if it is satisfiable. Decidability follows from the decidability of first-order theory of reals [44]. Towards this, we consider *primitive words*: a word w is *primitive* for S if a nice run of w in $\Gamma_{\mathcal{A}_S}$ has no cycles.

▶ **Theorem 20.** *It is decidable whether an FNFA is λ-resolvable.*

Proof Sketch. The key idea is that for each primitive word w and corresponding nice run π in the run automaton, we can obtain a number z_w, which is a lower bound on the probability of any word whose nice run is the same as π without the self loops. Moreover, we can find a sequence of such words whose limit probability is z_w. Hence, the automaton is λ-resolvable if and only if there is a support that satisfies $z_w \geq \lambda$ for every primitive word w.

To encode this idea in a formula in the theory of the reals, we have a variable x_τ^S for each $\tau \in S$, to express the probabilities assigned by some stochastic resolver over a given support S. Let Θ_S denote the set of constraints ensuring that the variables $\{x_\tau^S\}_{\tau \in S}$ satisfy the probabilistic constraints and take values strictly within $(0, 1]$.

The set P_S of all primitive words associated with the support S can be obtained from the run automaton $\Gamma_{\mathcal{A}}$, along with the corresponding symbolic probability z_w. We define the formula for the support S as $\varphi_S := \bigwedge_{w \in P_S} \varphi_w^S$, where $\varphi_w^S := z_w \geq \lambda$ for each $w \in P_S$. This gives our final system of inequalities $\varphi := \exists S \subseteq \Delta \; \exists \, \{x_\tau^S\}_{\tau \in S} (\Theta_S \wedge \varphi_S)$. ◀

We conclude by noting that our analysis does not extend to NFA that are not finitely-ambiguous, as Lemma 6 fails for bad supports in this setting.

7 Discussion

We have introduced the notion of λ-resolvability as a new measure for the degree of non-determinism. We showed that checking this property is undecidable for NFA, and provided further decidability and complexity results for important special cases, in particular finitely-ambiguous automata.

This opens new fields of study, especially whether there are more meaningful classes of NFA where positive or λ-resolvability are decidable. More immediate future work includes closing the remaining complexity gaps and discovering whether or not checking positive resolvability for general NFA is decidable.

Other challenges that arise are how these results extend to more expressive languages, such as (visible) counter or pushdown languages, and how they extend to ω-regular languages. For the latter challenge, several of our results transfer elegantly. The undecidability of λ-resolvability holds even for weak ω-automata. For finitely-ambiguous case, we can just adjust our definition of bad words to *bad prefixes*: where the accepting and rejecting runs in a bad word end in final and non-final states for bad words, for bad prefixes there needs to be some infinite word that is accepted from all states the prefixes of accepting runs end in but from none of the states the prefixes of non-accepting runs end in. Checking if such a word exists can be done in PSPACE for nondeterministic parity automata (NPA). It follows that the positive resolvability problem for finitely-ambiguous NPA is PSPACE-complete. The λ-resolvability problem is also decidable for finitely-ambiguous NPA. We give details in Appendix A.

References

1 Bader Abu Radi and Orna Kupferman. Minimizing gfg transition-based automata. In *International Colloquium on Automata, Languages and Programming (ICALP)*, 2019. doi:10.4230/LIPIcs.ICALP.2019.100.

2 Rajab Aghamov, Christel Baier, Toghrul Karimov, Joris Nieuwveld, Joël Ouaknine, Jakob Piribauer, and Mihir Vahanwala. Model checking markov chains as distribution transformers. In *Principles of Verification: Cycling the Probabilistic Landscape - Essays Dedicated to Joost-Pieter Katoen on the Occasion of His 60th Birthday, Part II*, volume 15261 of *Lecture Notes in Computer Science*, pages 293–313. Springer, 2024. doi:10.1007/978-3-031-75775-4_13.

3 Marc Bagnol and Denis Kuperberg. Büchi Good-for-Games Automata Are Efficiently Recognizable. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 16:1–16:14, 2018. doi:10.4230/LIPIcs.FSTTCS.2018.16.

4 Christel Baier, Luca de Alfaro, Vojtěch Forejt, and Marta Kwiatkowska. *Model Checking Probabilistic Systems*, pages 963–999. Springer International Publishing, 2018. doi:10.1007/978-3-319-10575-8_28.

5 Christel Baier, Florian Funke, Simon Jantsch, Toghrul Karimov, Engel Lefaucheux, Joël Ouaknine, David Purser, Markus A. Whiteland, and James Worrell. Parameter synthesis for parametric probabilistic dynamical systems and prefix-independent specifications. In *33rd International Conference on Concurrency Theory, CONCUR 2022, September 12-16, 2022, Warsaw, Poland*, volume 243 of *LIPIcs*, pages 10:1–10:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICS.CONCUR.2022.10.

6 Christel Baier, Holger Hermanns, and Joost-Pieter Katoen. *The 10,000 Facets of MDP Model Checking*, pages 420–451. Springer-Verlag, 2022. doi:10.1007/978-3-319-91908-9_21.

7 Christel Baier, Stefan Kiefer, Joachim Klein, David Müller, and James Worrell. Markov chains and unambiguous automata. *J. Comput. Syst. Sci.*, 136:113–134, 2023. doi:10.1016/J.JCSS.2023.03.005.

8 Udi Boker and Karoliina Lehtinen. Good for games automata: From nondeterminism to alternation. In *International Conference on Concurrency Theory (CONCUR)*, volume 140, pages 19:1–19:16, 2019. doi:10.4230/LIPIcs.CONCUR.2019.19.

9 Udi Boker and Karoliina Lehtinen. History Determinism vs. Good for Gameness in Quantitative Automata. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 38:1–38:20, 2021. doi:10.4230/LIPIcs.FSTTCS.2021.38.

10 Udi Boker and Karoliina Lehtinen. Token games and history-deterministic quantitative automata. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 120–139. Springer International Publishing, 2022. doi:10.1007/978-3-030-99253-8_7.

11 Sougata Bose, Thomas A. Henzinger, Karoliina Lehtinen, Sven Schewe, and Patrick Totzke. History-deterministic timed automata are not determinizable. In *International Workshop on Reachability Problems (RP)*, 2022. doi:10.1007/978-3-031-19135-0_5.

12 Sougata Bose, Thomas A. Henzinger, Karoliina Lehtinen, Sven Schewe, and Patrick Totzke. History-deterministic timed automata. *Log Meth Comput Sci*, Volume 20, Issue 4, October 2024. doi:10.46298/lmcs-20(4:1)2024.

13 Dmitry Chistikov, Stefan Kiefer, Andrzej S. Murawski, and David Purser. The Big-O Problem. *Log Meth Comput Sci*, 18(1), 2022. doi:10.46298/LMCS-18(1:40)2022.

14 Thomas Colcombet. The theory of stabilisation monoids and regular cost functions. In *International Colloquium on Automata, Languages and Programming (ICALP)*, pages 139–150, 2009. doi:10.1007/978-3-642-02930-1_12.

15 Wojciech Czerwinski, Engel Lefaucheux, Filip Mazowiecki, David Purser, and Markus A. Whiteland. The boundedness and zero isolation problems for weighted automata over nonnegative rationals. In *LICS '22: 37th Annual ACM/IEEE Symposium on Logic in Computer Science, Haifa, Israel, August 2 - 5, 2022*, pages 15:1–15:13. ACM, 2022. doi:10.1145/3531130.3533336.

16 Laure Daviaud, Marcin Jurdziński, Ranko Lazić, Filip Mazowiecki, Guillermo A. Pérez, and James Worrell. When are emptiness and containment decidable for probabilistic automata? *J. Comput. Syst. Sci.*, 119:78–96, 2021. doi:10.1016/j.jcss.2021.01.006.

17 Nathanaël Fijalkow, Hugo Gimbert, Edon Kelmendi, and Youssouf Oualhadj. Deciding the value 1 problem for probabilistic leaktight automata. *Log. Methods Comput. Sci.*, 11(2), 2015. doi:10.2168/LMCS-11(2:12)2015.

18 Nathanaël Fijalkow, Cristian Riveros, and James Worrell. Probabilistic automata of bounded ambiguity. *Information and Computation*, 282:104648, 2022. Special issue on 9th International Workshop Weighted Automata: Theory and Applications (WATA 2018). doi:10.1016/j.ic.2020.104648.

19 Hugo Gimbert and Youssouf Oualhadj. Probabilistic automata on finite words: Decidable and undecidable problems. In *Automata, Languages and Programming*, pages 527–538, 2010. doi:10.1007/978-3-642-14162-1_44.

20 Shibashis Guha, Ismaël Jecker, Karoliina Lehtinen, and Martin Zimmermann. A Bit of Nondeterminism Makes Pushdown Automata Expressive and Succinct. In *International Symposium on Mathematical Foundations of Computer Science (MFCS)*, pages 53:1–53:20, 2021. doi:10.4230/LIPIcs.MFCS.2021.53.

21 Ernst Moritz Hahn, Guangyuan Li, Sven Schewe, Andrea Turrini, and Lijun Zhang. Lazy probabilistic model checking without determinisation. In *International Conference on Concurrency Theory (CONCUR)*, pages 354–367, 2015. doi:10.4230/LIPIcs.CONCUR.2015.354.

22 Ernst Moritz Hahn, Mateo Perez, Sven Schewe, Fabio Somenzi, Ashutosh Trivedi, and Dominik Wojtczak. Good-for-mdps automata for probabilistic analysis and reinforcement learning. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 12078, pages 306–323. Springer, 2020. doi:10.1007/978-3-030-45190-5_17.

23 Thomas A. Henzinger, Karoliina Lehtinen, and Patrick Totzke. History-deterministic timed automata. In *International Conference on Concurrency Theory (CONCUR)*, 2022. doi:10.4230/LIPIcs.CONCUR.2022.14.

24 Thomas A. Henzinger and Nir Piterman. Solving games without determinization. In *Computer Science Logic (CSL)*, pages 395–410. Springer Berlin Heidelberg, 2006. doi:10.1007/11874683_26.

25 Thomas A. Henzinger, Aditya Prakash, and K. S. Thejaswini. Resolving nondeterminism with randomness, 2025.

26 Dexter Kozen. Lower bounds for natural proof systems. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 254–266, 1977. doi:10.1109/SFCS.1977.16.

27 Denis Kuperberg and Michał Skrzypczak. On determinisation of good-for-games automata. In *International Colloquium on Automata, Languages and Programming (ICALP)*, pages 299–310, 2015. doi:10.1007/978-3-662-47666-6_24.

28 Orna Kupferman, Shmuel Safra, and Moshe Y Vardi. Relating word and tree automata. *Ann. Pure Appl. Logic*, 138(1-3):126–146, 2006. doi:10.1016/J.APAL.2005.06.009.

29 Orna Kupferman and Moshe Y. Vardi. Weak alternating automata are not that weak. *ACM Trans. Comput. Log.*, 2(3):408–429, 2001. doi:10.1145/377978.377993.

30 Gregory F. Lawler. *Introduction to Stochastic Processes*. Chapman and Hall/CRC, 2nd edition, 2010.

31 Karoliina Lehtinen and Aditya Prakash. The 2-token theorem: Recognising history-deterministic parity automata efficiently, 2025.

32 Karoliina Lehtinen and Martin Zimmermann. Good-for-games ω-pushdown automata. *Log Meth Comput Sci*, 18, 2022. doi:10.1145/3373718.3394737.

33 Yong Li, Soumyajit Paul, Sven Schewe, and Qiyi Tang. Accelerating markov chain model checking: Good-for-games meets unambiguous automata. In *Computer Aided Verification (CAV)*, 2025.

34 J.R. Norris. *Markov Chains*. Cambridge University Press, 1998.

35 Alexander Okhotin. Unambiguous finite automata over a unary alphabet. *Information and Computation*, 212:15–36, 2012. doi:10.1016/j.ic.2012.01.003.

36 Joël Ouaknine and James Worrell. Positivity problems for low-order linear recurrence sequences. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2014, Portland, Oregon, USA, January 5-7, 2014*, pages 366–379. SIAM, 2014. doi:10.1137/1.9781611973402.27.

37 Azaria Paz. *Introduction to Probabilistic Automata*. Academic Press, 2014.

38 Alexander Rabinovich and Doron Tiferet. Degrees of ambiguity for parity tree automata. In Christel Baier and Jean Goubault-Larrecq, editors, *29th EACSL Annual Conference on Computer Science Logic, CSL 2021, January 25-28, 2021, Ljubljana, Slovenia (Virtual Conference)*, volume 183 of *LIPIcs*, pages 36:1–36:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICS.CSL.2021.36.

39 Walter J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4(2):177–192, 1970. doi:10.1016/S0022-0000(70)80006-X.

40 Sven Schewe. Büchi complementation made tight. In Susanne Albers and Jean-Yves Marion, editors, *26th International Symposium on Theoretical Aspects of Computer Science, STACS 2009, February 26-28, 2009, Freiburg, Germany, Proceedings*, volume 3 of *LIPIcs*, pages 661–672. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Germany, 2009. doi:10.4230/LIPICS.STACS.2009.1854.

41 Sven Schewe. Minimising good-for-games automata is NP-complete. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, 2020. doi:10.4230/LIPIcs.FSTTCS.2020.56.

42 Erik Meineche Schmidt. Succinctness of description of context-free, regular and unambiguous languages. *Cornell University*, 1978. doi:10.5555/908560.

43 L. J. Stockmeyer and A. R. Meyer. Word problems requiring exponential time (preliminary report). In *Symposium on Theory of Computing (STOC)*, pages 1–9. Association for Computing Machinery, 1973.

44 Alfred Tarski and J. C. C. McKinsey. *A Decision Method for Elementary Algebra and Geometry.* University of California Press, dgo - digital original, 1 edition, 1951. doi:10.2307/jj.8501420.

45 Mihir Vahanwala. Skolem and positivity completeness of ergodic markov chains. *Inf. Process. Lett.*, 186:106481, 2024. doi:10.1016/J.IPL.2024.106481.

46 Andreas Weber and Helmut Seidl. On the degree of ambiguity of finite automata. *Theor Comput Sci*, 88(2):325–349, 1991. doi:10.1016/0304-3975(91)90381-B.

A Application to Automata over Infinite Words

A *nondeterministic parity automaton* (NPA) $\mathcal{A}$ is a finite alphabet Σ, a finite set of states Q, an initial state q_0, a set of transitions $\Delta \subseteq Q \times \Sigma \times Q$, a priority function $\rho : Q \mapsto [d]$ for some $d \in \mathbb{N}$. An NPA accepts a set of infinite words in Σ^ω. A run π of $\mathcal{A}$ on an infinite word $w = \sigma_1\sigma_2 \cdots \in \Sigma^\omega$ is an infinite sequence of transitions $\tau_1\tau_2 \ldots$, where each $\tau_i = (p_i, \sigma_i, q_i)$ for $i \in \mathbb{Z}_+$. A run π naturally generates an infinite trace $\rho(\pi)$ over $[d]$, where the i-th element in this sequence is $\rho(p_i)$. Let $inf_\rho(\pi)$ be the numbers that occur infinitely often in $\rho(\pi)$. The run π is accepting if $p_1 = q_0$ and the maximum element in $inf_\rho(\pi)$ is even. A Büchi automaton is an NPA where ρ takes values in $\{1, 2\}$. A Büchi automaton can be described by providing the set of states with priority 2 as the accepting set of states. A Büchi automata is weak iff every SCC contains either only accepting states or only rejecting states. Let $\mathrm{ACC}_\mathcal{A}(w)$ denote the set of all accepting runs of $\mathcal{A}$ on w. The notions of k-ambiguous and finitely-ambiguous naturally extends to NPA based on the cardinality of $\mathrm{ACC}_\mathcal{A}(w)$.

A word $w \in \Sigma^\omega$ induces a probability measure $Prob_w^\mathcal{P}$ over the measurable subsets of Δ^ω, defined via the σ-algebra generated by basic cylinder sets in the standard way. The acceptance probability of w in $\mathcal{P}$ is then given by:

$$\mathcal{P}(w) = Prob_w^\mathcal{P}(\mathrm{ACC}(w)).$$

Definitions of positive resolvability and λ-resolvability extends naturally to NPA.

A.1 Undecidability of λ-Resolvability

We have shown that the λ-resolvability problem for automata over finite words (i.e., NFA) is undecidable. In this subsection, we extend this result to automata over infinite words.

Given an NFA $\mathcal{A} = (\Sigma, Q, q_0, F, \Delta)$, define the Büchi automaton $\mathcal{A}_\omega = (\Sigma', Q', q_0, F', \Delta')$ as follows:

- $\Sigma' = \Sigma \cup \{\sharp\}$, where $\sharp \notin \Sigma$.
- $Q' = Q \cup \{q'_f\}$, where $q'_f \notin Q$.
- $F' = \{q'_f\}$.
- $\Delta' = \Delta \cup \{(p, \sharp, q'_f) \mid p \in F \cup F'\}$.

By construction, $\mathcal{A}_\omega$ has a unique accepting state q'_f, which is also a sink state, making $\mathcal{A}_\omega$ a weak automaton. Suppose $\mathcal{P}$ is a PFA based on $\mathcal{A}$. We construct the corresponding probabilistic automaton $\mathcal{P}_\omega$ based on $\mathcal{A}_\omega$ by adding transitions of the form $(p, \sharp, q'_f, 1)$ to $\mathcal{P}_\omega$ for all $p \in F \cup F'$. This construction ensures that $\mathcal{P}(w) = \mathcal{P}_\omega(w \cdot \sharp^\omega)$ for every word $w \in \Sigma^*$. Consequently, for every $\lambda \in (0, 1)$, $\mathcal{A}$ is λ-resolvable if and only if $\mathcal{A}_\omega$ is λ-resolvable. Together with the undecidability of the λ-resolvability problem for NFA (Theorem 2), we obtain the following:

▶ **Theorem 21.** *The λ-resolvability problem for weak nondeterministic Büchi automata is undecidable.*

A.2 Decidability for Finitely-ambiguous NPA

Our algorithm for FNFA also extends to finitely-ambiguous NPA using similar analysis. Since the number of accepting runs of every word is bounded by some k, Lemma 6 holds for finitely-ambiguous NPA as well.

▶ **Lemma 22.** *Let S be a support for a finitely-ambiguous NPA $\mathcal{A}$ with $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}_S)$. Then S is a bad support if and only if for every $b \in \mathbb{N}$, there is at least one word $w \in \mathcal{L}(\mathcal{A}_S)$ such that w has at least b nondeterministic transitions from S in each of its accepting runs.*

Hence it is enough to track the behaviour of the NPA up to a finite prefix. Towards this goal, the notion of bad words can be adapted to NPA using *bad prefixes*. Let $\mathcal{A}^q$ denote the ω-automaton $\mathcal{A}$ with q as its initial state.

▶ **Definition 23** (Bad prefix). *Let $\mathcal{A}$ be a finitely-ambiguous NPA and $S \subseteq \Delta$ be a support. Then a word $u \in \Sigma^*$ is a bad prefix for S if for some ℓ, there exists a decomposition $u = x_0 y_1 x_1 \dots y_\ell$ with words $x_0, \dots, x_{\ell-1} \in \Sigma^*$, $y_1, \dots, y_\ell \in \Sigma^+$, sets of states $Q_1, \dots, Q_\ell, R_1, \dots, R_\ell \subseteq Q$, states $q_1, \dots, q_\ell$ with $q_i \in Q_i$ for each $i \in [\ell]$ and an ω-word $v \in \Sigma^\omega$ such that $uv \in \mathcal{L}(\mathcal{A}_S)$ and*

1. *For each $i \in [\ell]$, all accepting runs of uv from q_0 end in Q_i after reading $x_0 y_1 \dots x_{i-1}$, i.e., $q \in \delta_{\mathcal{A}_S}(q_0, x_0 y_1 \dots x_{i-1})$ and $y_i \dots y_\ell v \in \mathcal{L}(\mathcal{A}_S^q) \iff q \in Q_i$.*
2. *An accepting run of uv must be in some q_i for some $i \in [\ell]$ after reading $x_0 y_1 x_1 \dots y_i$.*
3. *For each $i \in [\ell]$ and each $q \in Q_i$, every accepting run of uv that reaches q after reading the prefix $x_0 y_1 \dots x_{i-1}$ also returns to q after subsequently reading y_i. Moreover, when $q = q_i$, this run of y_i from q_i to q_i contains at least one transition from S which is nondeterministic in $\mathcal{A}_S$.*
4. *For each $i \in [\ell]$, R_i contains all states that are reached from q_0 after reading $x_0 y_1 \dots x_{i-1}$, but has no continuation to an accepting run for the remaining suffix $y_i x_i \dots y_\ell v$, i,e., $q \in \delta_{\mathcal{A}_S}(q_0, x_0 y_1 \dots x_{i-1})$ and $y_i \dots y_\ell v \notin \mathcal{L}(\mathcal{A}_S^q) \iff q \in R_i$. Moreover, R_i is also exactly*

the set of states that are reached from q_0 after reading $x_0y_1 \dots x_{i-1}y_i$, with no continuation to an accepting run for the remaining suffix $x_i \dots y_\ell v$, i.e., $q \in \delta_{\mathcal{A}_S}(q_0, x_0y_1 \dots x_{i-1}y_i)$ and $x_i \dots y_\ell v \notin \mathcal{L}(\mathcal{A}_S^q) \iff q \in R_i$.

▶ **Lemma 24** (Bad support-bad prefix)**.** *Let $\mathcal{A}$ be a finitely-ambiguous NPA and S be a support of $\mathcal{A}$. Then S is a bad support for $\mathcal{A}$ iff $\mathcal{L}(\mathcal{A}_S) \subsetneq \mathcal{L}(A)$ or there exists a bad prefix for S.*

Hence, our PSPACE algorithm for deciding the positive resolvability of FNFA can be extended to finitely-ambiguous NPA by adding an *accepting mode*. This mode verifies whether there exists an ω-word v in the set $\bigcap_{q\in A} \mathcal{L}(\mathcal{A}_S^q) \setminus \bigcup_{q\in R} \mathcal{L}(\mathcal{A}_S^q)$. In the branching mode, a transition to the accepting mode is always possible when $G = \emptyset$. Once the system enters the accepting mode, it remains there.

To check whether or not $\bigcap_{q\in A} \mathcal{L}(\mathcal{A}_S^q) \setminus \bigcup_{q\in R} \mathcal{L}(\mathcal{A}_S^q)$ is empty, we check if $\bigcap_{q\in A} \mathcal{L}(\mathcal{A}_S^q) \subseteq \bigcup_{q\in R} \mathcal{L}(\mathcal{A}_S^q)$ holds. To do this, we can simply translate the individual $\mathcal{A}_S^q$ for all $q \in A$ to NBAs $\mathcal{N}_q$ and construct an NBA $\mathcal{N}$ recognising $\bigcup_{q\in R} \mathcal{L}(\mathcal{A}_S^q)$; the blow-up for this is small.

We can now check whether or not $\bigcap_{q\in A} \mathcal{L}(\mathcal{N}_q) \subseteq \mathcal{L}(\mathcal{N})$ holds.

For this, we complement $\mathcal{N}$ using [40] to the complement NBA $\mathcal{C}$, but note that this complement – including the states and transitions – can be represented symbolically by $\mathcal{N}$ itself – it only uses subsets of the states in $\mathcal{N}$ (the reachable states and a subset thereof used for a breakpoint construction) and level rankings, which are functions from the reachable states of $\mathcal{N}$ to natural numbers up to $2n$, where n is the number of states of $\mathcal{N}$ [29, 40].

We can now check whether or not $\bigcap_{q\in A} \mathcal{L}(\mathcal{N}_q) \cap \mathcal{L}(\mathcal{C}) = \emptyset$ holds.

To do this, we can search in the exponentially larger, but polynomially represented, product space of $\bigotimes_{q\in A} \mathcal{N}_q \times \mathcal{C}$ (the product automaton of $\mathcal{N}_q$ for all $q \in A$ and $\mathcal{C}$) to check if we can reach a product state, which can loop to itself while seeing an accepting state of each of the automata. This can be checked in NL in the explicit size of the product automaton and thus in PSPACE in its representation.

▶ **Theorem 25.** *The positive resolvability problem for finitely-ambiguous NPA is* PSPACE*-complete.*

Let the degree of ambiguity of a finitely-ambiguous NPA $\mathcal{A}$ be k, which can be decided [38, Proposition 21]. Note that in [38], the notion of finite ambiguity differs from ours: an automaton is called finitely ambiguous if $|\operatorname{ACC}(w)|$ is finite for every word w in the language. What we refer to as finitely-ambiguous is instead termed *boundedly ambiguous* in their terminology.

Let S be a support of $\mathcal{A}$, we can construct a run automaton for $\mathcal{A}_S$, similar to the construction for an NFA on a support. As in the NFA case, by analysing the run automaton, we can build a finite system of inequalities and decide the λ-resolvability problem for finitely-ambiguous NPA.

▶ **Theorem 26.** *The λ-resolvability problem for finitely-ambiguous NPA is decidable.*

Chance and Mass Interpretations of Probabilities in Markov Decision Processes

Yun Chen Tsai
National Institute of Informatics, Tokyo, Japan
SOKENDAI (The Graduate University for Advanced Studies), Kanagawa, Japan

Kittiphon Phalakarn
National Institute of Informatics, Tokyo, Japan

S. Akshay
Department of CSE, Indian Institute of Technology Bombay, Mumbai, India

Ichiro Hasuo
National Institute of Informatics, Tokyo, Japan
SOKENDAI (The Graduate University for Advanced Studies), Kanagawa, Japan
Imiron Co., Ltd., Tokyo, Japan

Abstract

Markov decision processes (MDPs) are a popular model for decision-making in the presence of uncertainty. The conventional view of MDPs in verification treats them as state transformers with probabilities defined over sequences of states and with schedulers making random choices. An alternative view, especially well-suited for modeling dynamical systems, defines MDPs as distribution transformers with schedulers distributing probability masses. Our main contribution is a unified semantical framework that accommodates these two views and two new ones. These four semantics of MDPs arise naturally through identifying different sources of randomness in an MDP (namely *schedulers*, *configurations*, and *transitions*) and providing different ways of interpreting these probabilities (called the *chance* and *mass* interpretations). These semantics are systematically unified through a mathematical construct called *chance-mass (CM) classifier*. As another main contribution, we study a reachability problem in each of the two new semantics, demonstrating their hardness and providing two algorithms for solving them.

2012 ACM Subject Classification Theory of computation → Random walks and Markov chains; Theory of computation → Verification by model checking

Keywords and phrases MDP, distribution transformer, antichain, template-based synthesis

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2025.33

Related Version *Full Version*: https://arxiv.org/abs/2506.10377

Funding The authors are supported by ASPIRE Grant No. JPMJAP2301, and ERATO HASUO Metamathematics for Systems Design Project (No. JPMJER1603), JST. S.A. is supported by the SBI Foundation Hub for Data Science & Analytics, IIT Bombay. I.H. is supported by CREST ZT-IoT Project (No. JPMJCR21M3), JST.

1 Introduction

MDPs and Conventional Semantics. Markov decision processes (MDPs) are a classical model combining nondeterminism and randomness, with a rich theory and practical uses in a wide variety of applications [26]. MDPs are often viewed as state transformers where a move from one state to another is defined as follows: from a given state, an action is chosen by a scheduler out of multiple possible actions and then a probabilistic transition is made leading to a resulting state. In other words, a scheduler resolves the nondeterminism in the MDP to give rise to a Markov chain (MC), a purely probabilistic model. In general, a scheduler can also be randomized (also called a mixed scheduler), and thus the actions are chosen according to the probabilities it defines. See [9] for more discussion.

© Yun Chen Tsai, Kittiphon Phalakarn, S. Akshay, and Ichiro Hasuo;
licensed under Creative Commons License CC-BY 4.0

MDPs as Distribution Transformers. A different semantics of MDPs (and Markov chains), inspired by dynamical systems, considers them as *distribution transformers*. In this view, we start with a probability distribution or mass over the set of all states of the MDP, and according to the action chosen by a scheduler, this mass gets re-distributed over the states. Therefore, a one-step transition results in a new probability distribution over the set of states, in which sense the MDP is a linear transformer of distributions. This semantics is studied in multiple recent works [1,2,4–6,10,11,19,21] with applications ranging from AI, where swarms of robots are modeled, to biological systems, where concentrations of biochemical reagents or populations of cells are modeled. This semantics is also common, e.g., in statistics and the theory of dynamical systems, where mathematicians are often interested in *stationary distributions*, which are essentially fixed points of the distribution transformer.

Reachability Problems in MDPs. The *reachability problem* is fundamental in model checking. In the context of MDPs, it asks whether there exists a scheduler such that the MDP, following this scheduler from a given state, reaches a given *target state* with a certain probability. It is known that the reachability problem for MDPs can be solved in polynomial time, and well-established probabilistic model checkers such as PRISM [20] and Storm [17] can handle large instances efficiently.

However, it has been observed that when the reachability problem on MDPs is "lifted" and is asked under the distribution transformer semantics, it becomes significantly harder. The "lifted" problem asks for the reachability probability to a *target distribution*, instead of to a target state. The problem is significantly harder even for Markov chains; in particular, it is as hard as the Skolem problem, a long-standing open problem on linear recurrences [3,23,24].

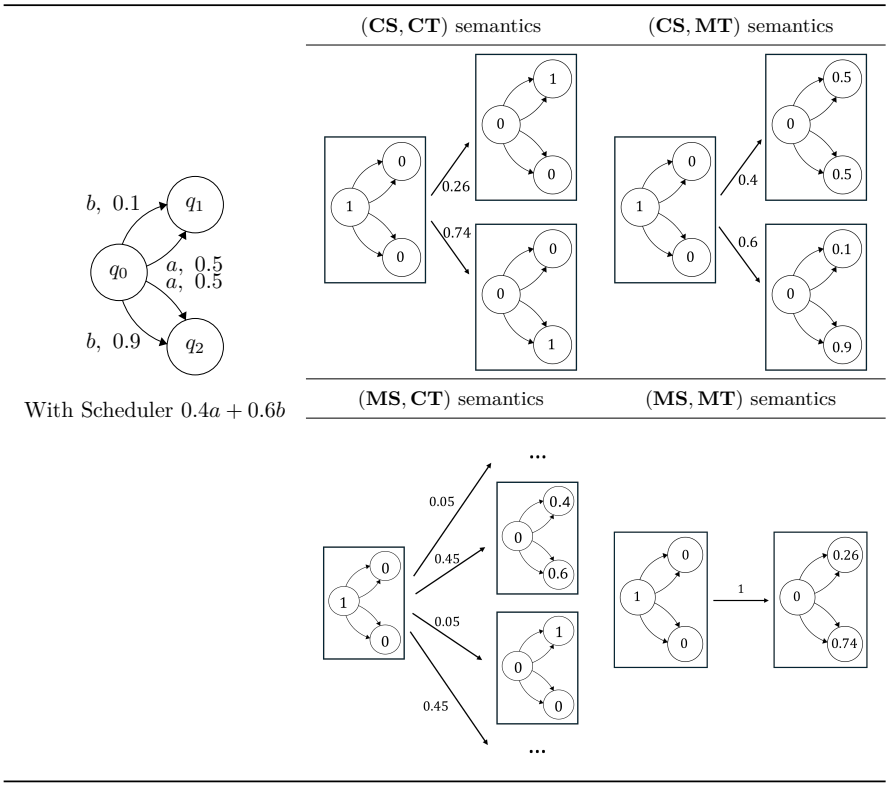
Motivating from these observations, we aim to develop a systematic framework that describes the two semantics and the "lifted" reachability problem. This leads to our main contribution: a unified framework for MDPs semantics.

Four Semantics for MDPs: a Unified Framework. We start by observing that there are two sources of probabilities in MDPs: those arising from randomized/mixed schedulers (such as 0.4 and 0.6 in Table 1) and the inherent transition probabilities (such as 0.1 and 0.9 in Table 1). On the other hand, these probabilities can be interpreted in two ways: the conventional manner, which we call the *chance interpretation*, models probabilities as random events like coin tosses; and the "distribution transformer" manner, which we call the *mass interpretation*, models the probabilities as the rate of splitting a mass of particles.

It is then natural to consider four variations of MDP semantics, where each of these two classes of probabilities – coming from the above two sources – is interpreted under either the *chance* or *mass* interpretation (cf. Table 1). For example, the conventional semantics corresponds to applying the *chance* interpretation to both scheduler and transition probabilities, as both schedulers and transitions are modeled as random events (like coin tosses). Similarly, the distribution transformer semantics corresponds to applying the *mass* interpretation to both scheduler and transition probabilities.

Let us explain the illustration in Table 1 in more detail. The left side of Table 1 shows a simple MDP with three states and two possible actions from q_0, where each action leads to a probabilistic transition over states q_1 and q_2. A mixed scheduler is fixed, which chooses action a with probability 0.4 and action b with probability 0.6. Each of the four Markov chains (MCs) shown on the right side of Table 1, corresponding to each semantics, is called the *config MC* for the MDP and the specified scheduler. The intuition here is that these config MCs are Markov chains where each state represents a mass distribution over the state space of the MDP on the left, and they model how mass distributions over the MDP states evolve under the specified interpretation of probabilities.

Table 1 Examples of the four semantics of MDPs.



Let's take the $(\mathbf{CS}, \mathbf{MT})$, an abbreviation for **C**hance-interpreted **S**cheduler, **M**ass-interpreted **T**ransitions, semantics as an example. As suggested by its naming, this semantics interprets the probability from the scheduler as a random event, and the probability from the transition as a rate of splitting mass. In one step of the MDP, an (unfair) coin is first tossed to decide the action to be taken by the scheduler: a 0.4 chance of choosing action a and a 0.6 chance of choosing action b. This corresponds to the two transitions of the config MC as illustrated in Table 1. Once the action is determined, the mass of 1 at q_0 is then split according to the transition probabilities, which yields a distribution of 0.5 at q_1 and 0.5 at q_2 in the case of action a (the 0.4 branch). The process for the case of action b is similar. These two distributions are the only distributions that can be reached by the MDP after one step if the probabilities are interpreted as specified.

Table 1 also clearly demonstrates that the reachability problem has different answers under different semantics. For example, the mass distribution "0.26 at q_1, 0.74 at q_2" is reached only under the $(\mathbf{MS}, \mathbf{MT})$ semantics; the mass distribution "1 at q_1" is reached with the probability 0.27 under the $(\mathbf{CS}, \mathbf{CT})$ semantics, while it is with the probability 0.05 under the $(\mathbf{MS}, \mathbf{CT})$ semantics.

Examples. All four semantics are not only theoretically natural, but also lead to different application scenarios. For the purely random $(\mathbf{CS}, \mathbf{CT})$ semantics, examples are abundant in the CS literature [9], and similarly for the purely distributional $(\mathbf{MS}, \mathbf{MT})$ semantics, examples are found in the dynamical systems and AI literature [5].

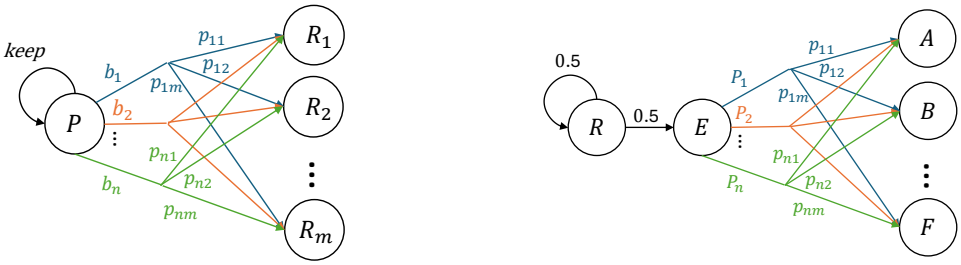


Figure 1 *Casino* MDP for $(\mathbf{MS}, \mathbf{CT})$ sem. (left); *exam* MDP for $(\mathbf{CS}, \mathbf{MT})$ sem. (right).

Casino. An example suited for the $(\mathbf{MS}, \mathbf{CT})$ semantics is the *casino* MDP in Fig. 1. Here, P is the initial state, from which actions b_i ($i \in \{1, \dots, n\}$, "bet at Game i") and keep ("skip this round") are offered. Each bet b_i can lead to various rewards – represented by states R_j – and the probabilities p_{ij} are known. What is notable about the $(\mathbf{MS}, \mathbf{CT})$ semantics is that a mixed scheduler $r_{\mathsf{keep}}\mathsf{keep} + r_1 b_1 + \cdots + r_n b_n$ nicely models a gambler's strategy to split the fund for different games (betting the r_1 portion of the fund at Game 1, the r_2 portion at Game 2, etc.). In contrast, the transition probabilities p_{ij} should be interpreted in the chance, not mass, manner. This is because the rewards from each game are random events and only one outcome is produced at each game i in each round (for example you either win or lose with a slot machine, instead of getting a half-and-half outcome). Another point illustrated here is that the scheduler can choose its action depending on the current mass. For example, a possible strategy is to spare half the original fund for the second round (using a strategy $\frac{1}{2}\mathsf{keep} + \cdots$), and to decide how to use it depending on the outcome of the first round (i.e., the mass over the states $R_1, \dots, R_m$ after one transition).

Exam. As an example suited for the $(\mathbf{CS}, \mathbf{MT})$ semantics, we present the *exam* MDP in Fig. 1. Here, one chooses a problem set P_i from a given repertoire $P_1, \dots, P_n$, as commonly done in language tests such as TOEFL and IELTS. Difficulties can differ among problem sets: with the problem set P_i, the p_{ij} portion of the examinees gets the grade $j \in \{A, B, \dots, F\}$, and this number p_{ij} is known. Exams are repeated, and different problem sets can be used at each time. The state R with its transitions models that the mass of potential examinees decays exponentially over time. Here, a mixed scheduler $r_1 P_1 + \cdots + r_n P_n$ models the strategy that the problem set P_i is chosen with the probability r_i. Each time, one uses the same problem set for all examinees, so r_i should be chance-interpreted. In contrast, the transition probabilities p_{ij} should be mass-interpreted: every individual has a chance of getting a certain grade while the overall distribution over grades should follow the distribution indicated by the paper's difficulties.

Contributions. In this paper, we are interested in 1) formulating a unified framework that captures all four semantics and 2) solving the *threshold reachability* problem under the new semantics we introduce. Given an MDP $\mathcal{M}$ and a scheduler σ, choosing a semantics (out of the four) induces the so-called *config MC* $\mathcal{M}_\sigma$; then the threshold reachability problem asks if a target set H can be reached with probability greater than a given threshold ξ. As mentioned earlier, this problem is poly-time solvable for the $(\mathbf{CS}, \mathbf{CT})$ semantics, while being Skolem hard for the $(\mathbf{MS}, \mathbf{MT})$ semantics. Hence, it is natural to ask how hard it is for the new $(\mathbf{CS}, \mathbf{MT})$ and $(\mathbf{MS}, \mathbf{CT})$ semantics, and if we can develop approaches to solve it.

Our contributions (as summarized in Table 2) are as follows.

1. A unified framework where different semantics for MDPs are systematically derived by appropriately specifying the so-called *Chance-Mass (or CM) classifier*.

Table 2 Summary of semantics/results: Blue are our contributions, Grey were previously known.

		Transition	
		Chance-interpreted	**M**ass-interpreted
Scheduler	**C**hance-interpreted	(**CS**, **CT**) semantics Ptime	(**CS**, **MT**) semantics Undecidable, antichain algo
	Mass-interpreted	(**MS**, **CT**) semantics #P-hard, template algo	(**MS**, **MT**) semantics Skolem-hard

2. For the (**CS**, **MT**) semantics, we show that this problem is undecidable under a global scheduler and we build an antichain-based backward reachability algorithm for finite union of finitely-generated monotone sets that is sound but necessarily incomplete.
3. For the (**MS**, **CT**) semantics, we show the problem is #P-hard – the counting counterpart of NP, see e.g. [7] – even for a single action in the MDP, and we develop a template-based submartingale which provides a sound algorithm for reachability.
4. We implemented both these algorithms in a prototype. This illustrates the working of our algorithms on a small set of examples including the motivating examples above.

Related Work. Beyond the related work already discussed, here we add some more context and mention a few others. First, the use of distributional (**MS**, **MT**) semantics for MDPs has been long studied, but mostly from a theoretical point of view. See e.g., [19]. Recently, a template-based approach is implemented in [4] for safety constraints and [5] for reach-avoidance, where the authors use Farkas Lemma [12] and Handelman's [16] theorem to develop sound but incomplete algorithms via synthesizing ranking functions. For the (**CS**, **MT**) semantics, we use γ-scaled submartingales for reachability as done e.g., in [27]. Antichain algorithms have been developed for LTL synthesis e.g., in [13], but to the best of our knowledge, we are the first to use them in the context of MDPs.

2 Preliminaries

We let $\mathbb{N}, \mathbb{Q}, \mathbb{R}$ denote the sets of natural numbers, rational numbers and real numbers, respectively. The unit interval is denoted by $[0,1]$, and we let $[n] = \{1, 2, \ldots, n\}$. For sets A and B, 2^A denotes the powerset of A, and B^A denotes the set of functions $f : A \to B$. The collection of all finite (or infinite) sequences over A is denoted by A^* (or A^ω, respectively). $A^+ \subseteq A^*$ is the set of nonempty finite sequences.

Probability Distributions. For any set X, we let $\mathcal{D}(X) = \{d \colon X \to [0,1] \mid \sum_x d(x) = 1, \mathrm{supp}(d) \text{ is finite}\}$ denote the set of *finitely-supported discrete probability distributions* over X. Here, $\mathrm{supp}(d) = \{x \in X \mid d(x) > 0\}$ is the *support* of d. In this paper, we call $d \in \mathcal{D}(X)$ simply a *distribution* over X.

An n-dimensional *stochastic matrix* is a $n \times n$ matrix $M \in [0,1]^{n \times n}$ such that $M\mathbf{1} = \mathbf{1}$. We let $\mathsf{Stoc}(n)$ denote the set of n-dimensional stochastic matrices.

Markov Decision Processes. A *Markov decision process* (MDP) $\mathcal{M} = (Q, \mathsf{Act}, \delta)$ consists of a finite set Q of *states*, a finite set Act of *actions*, and a *transition function* $\delta : Q \times \mathsf{Act} \to \mathcal{D}(Q)$. We write M_a for the *transition matrix* of an MDP $\mathcal{M}$ for an action a; concretely $(M_a)_{ij} = \delta(q_i, a)(q_j)$. Using this, we also write $\mathcal{M} = (Q, \mathsf{Act}, (M_a)_{a \in \mathsf{Act}})$ for the same MDP.

A distribution over Q, i.e. $d \in \mathcal{D}(Q)$, shall be called a *configuration* of the MDP $\mathcal{M}$. This terminology is because of their use in Sec. 3.1.

Antichains and Monotone Sets. For a partially-ordered set $(X, \leq)$, a subset $S \subseteq X$ is an *antichain* if for any $x, y \in S$, $x \not\leq y$ and $y \not\leq x$. A subset $S \subseteq X$ is *upward closed* if for any $x \in S$ and $y \in X$, $x \leq y$ implies $y \in S$. For a upward-closed set $S \subseteq X$, the *bottom* of S is defined by $\lfloor S \rfloor = \{x \in S \mid \forall y \in S, y \leq x \implies x = y\}$; this collects all $\leq$-minimal elements of S and is clearly an antichain. Given an arbitrary subset $T \subseteq X$, its upward closure $\uparrow T = \{x \in X \mid \exists y \in T, y \leq x\}$ is upward-closed. We say an upward-closed set $S \subseteq X$ is *finitely generated* if $S = \uparrow T$ for some finite set T. We do not lose generality if we restrict this T to be an antichain. In this case, we have $\lfloor S \rfloor = T$. The dual notions, namely of *downward-closed set* and its *top*, are defined similarly. A set is *monotone* if it is either upward-closed or downward-closed.

3 A Unified Framework for Mass and Chance Interpretations

Our unified framework translates an MDP $\mathcal{M}$ and a scheduler σ to the so-called *configuration Markov chain (config MC)* $\mathcal{M}_\sigma$; the last is the semantic basis on which we can ask problems such as reachability. The transition function of the config MC $\mathcal{M}_\sigma$ is defined systematically as the composition $\mathbb{X} \circ \delta_\sigma$ of a *chance-mass classifier (CM classifier)* $\mathbb{X}$ and a function δ_σ; see Def. 9. There is a notable separation of concerns here: a CM classifier $\mathbb{X}$ specifies which semantics we use (cf. Table 2) but is independent of $\mathcal{M}$ or σ (Def. 7); the function δ_σ (Def. 6), in contrast, is defined by $\mathcal{M}$ (whose transition function is δ) and a scheduler σ, but is independent of the choice of the semantics.

The following basic constructs and notations will be used throughout the construction. The notation $\mathcal{D}g$ – inspired by category theory – may be nonstandard but is useful in Sec. 4.

▶ **Notation 1** (currying and pushforward). *Let X, Y, Z be sets. The* currying *of $f \colon X \times Y \to Z$ is denoted by $f^\wedge \colon X \to Z^Y$. For $g \colon X \to Y$, the function $\mathcal{D}g \colon \mathcal{D}(X) \to \mathcal{D}(Y)$ carries a (discrete) distribution $d \in \mathcal{D}(X)$ to the* pushforward measure $\big(\mathcal{D}g(d)\big)(y)$ *given by* $\big(\mathcal{D}g(d)\big)(y) = \sum_{x \in X, g(x)=y} d(x)$.

▶ **Notation 2** (ket notation). *For distributions, we use the following* ket notation*: for $d \in \mathcal{D}(X)$, we write $d = \sum_x a_x |x\rangle$ with $a_x = d(x) \in [0, 1]$. Intuitively, if we consider $d = \sum_x a_x |x\rangle$ as a random variable, a_x is the probability of obtaining the value x. More generally, we may use an arbitrary index set I and a function $f \colon I \to X$ and write $\sum_{i \in I} a_i |f(i)\rangle$. This distribution assigns to each $x \in X$ the probability $\sum_{i \in I, f(i)=x} a_i$.*

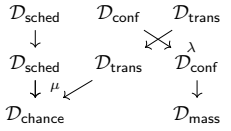
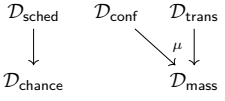
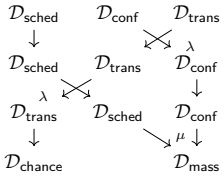
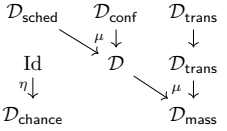
We note two special cases of the usages of the ket notation. Firstly, when $I = X' \subseteq X$ is a subset of X, the distribution $d = \sum_{x \in X'} a_x |x\rangle$ assigns 0 by case distinction: $d(x) = a_x$ if $x \in X'$, and $d(x) = 0$ otherwise. Secondly, the pushforward measure $\mathcal{D}f(d)$, with $d = \sum_{x \in X} a_x |x\rangle$ and $f \colon X \to Y$, can be conveniently denoted by $\mathcal{D}f(d) = \sum_{x \in X} a_x |f(x)\rangle$.

3.1 CM Classifiers, Pre-configurations, and Config Markov Chains

We introduce the central constructs of our framework, under two different views:

- the *concrete* and *element-level* view, describing the constructs as functions; and
- the *abstract* and *type-level* view, providing high-level intuitions of the nature of each construct using its type.

Table 3 Combinatorial presentation of CM classifiers $\mathbb{X}$.

$\mathbb{X}^{(\mathbf{CS},\mathbf{CT})}$	$\mathbb{X}^{(\mathbf{CS},\mathbf{MT})}$	$\mathbb{X}^{(\mathbf{MS},\mathbf{CT})}$	$\mathbb{X}^{(\mathbf{MS},\mathbf{MT})}$
			

The latter view is in fact that of *category theory* [8,18,22], but its precise treatment is beyond the scope of this paper.

We highlight the role of the type-level view through a preview of one of our main constructs, namely a *CM classifier*. It is formulated as a function $\mathbb{X}_Q \colon \mathcal{D}(\mathcal{D}(\mathcal{D}(Q))) \to \mathcal{D}(\mathcal{D}(Q))$. The intuition of a CM classifier $\mathbb{X}_Q$, in the abstract type-level view, is as in the following preview. (The preview focuses on type-level intuitions and is not meant to be understood at the first look – it is really a *preview*. Readers can skim through it and come back.)

- In our framework for MDP semantics, there are three sources of probability distributions (namely *schedulers*, *configurations*, and *transitions*); they are represented by the three occurrences of the distribution operator $\mathcal{D}$ in the domain of $\mathbb{X}_Q$.
- Each of the three kinds of probabilities are interpreted either in a *chance* or *mass* manner; these two manners are represented by the two occurrences of $\mathcal{D}$ in the codomain of $\mathbb{X}_Q$.
- For intuition, we often annotate $\mathcal{D}$'s with their intentions, namely $\mathcal{D}_{\mathsf{sched}}, \mathcal{D}_{\mathsf{conf}}, \mathcal{D}_{\mathsf{trans}}$ (for probabilities of different sources) and $\mathcal{D}_{\mathsf{chan}}, \mathcal{D}_{\mathsf{mass}}$ (for how they are interpreted). In particular, following the above intuitions, a CM classifier $\mathbb{X}_Q$ will have the type

$$\mathbb{X}_Q \colon \mathcal{D}_{\mathsf{sched}}(\mathcal{D}_{\mathsf{conf}}(\mathcal{D}_{\mathsf{trans}}(Q))) \longrightarrow \mathcal{D}_{\mathsf{chan}}(\mathcal{D}_{\mathsf{mass}}(Q)). \tag{1}$$

  Note that these $\mathcal{D}$'s have the same mathematical content regardless of the annotations.
- The function $\mathbb{X}_Q$ specifies which kind of probability (on the left) is interpreted in which manner (on the right). One combinatorial way of presenting such a specification $\mathbb{X}_Q$ is as a mapping from the three $\mathcal{D}$'s in the domain to the two $\mathcal{D}$'s in the codomain.
- Furthermore, such a mapping can be described by 1) swapping the order of $\mathcal{D}$'s, 2) suppressing two $\mathcal{D}$'s into one, and 3) inserting one $\mathcal{D}$ where there was none. See Table 3. We introduce functions λ, μ, η for conducting these operations (Def. 11); examples of $\mathbb{X}_Q$ are thus described as combinations of λ, μ, η.

We move on to the precise technical development, which we mostly do in the concrete, element-level view, with some type-level intuitions supplementing it. More examples of each component introduced in this section is given in [28].

In our general framework, a configuration of an MDP is a distribution over states.

▶ **Definition 3** (configuration, scheduler). *A* configuration *of an MDP $\mathcal{M} = (Q, \mathsf{Act}, \delta)$ is a distribution $d \in \mathcal{D}(Q)$. A* (configuration-based) scheduler *for $\mathcal{M}$ is a function $\sigma \colon \mathcal{D}(Q)^+ \to \mathcal{D}(\mathsf{Act})$ that maps a sequence of configurations to a distribution over actions. A scheduler is* memoryless *if its value only depends on the last element of its input, that is for any sequence $d_1 d_2 \ldots d_n \in \mathcal{D}(Q)^+$, $\sigma(d_1 d_2 \ldots d_n) = \sigma(d_n)$. Therefore, a memoryless scheduler is a function of the type $\sigma \colon \mathcal{D}(Q) \to \mathcal{D}(\mathsf{Act})$. A scheduler is* pure *if it always gives a Dirac distribution, that is for any $p \in \mathcal{D}(Q)^+$, $\sigma(p)(a) = 1$ for some $a \in \mathsf{Act}$. Otherwise, the scheduler is* mixed.

▶ Remark 4. Note that the scheduler notion in Def. 3 uses *global* actions: it chooses an action a is picked from the distribution $\sigma(d_1 d_2 ... d_n) \in \mathcal{D}(\mathsf{Act})$ and *this action a is used at every state $q \in Q$*. The more common notion of *local* action – allowing different actions at different states – can be encoded as global actions by bloating the action set.

A CM classifier (as presented in (1)) takes, as input, three-fold probability distributions, and its layers correspond to the three sources of probability in MDPs. We shall call such data a pre-configuration.

▶ **Definition 5** (pre-configuration). *Let $\mathcal{M} = (Q, \mathsf{Act}, \delta)$ be an MDP. A* pre-configuration *of $\mathcal{M}$ is an element $t \in \mathcal{D}(\mathcal{D}(\mathcal{D}(Q)))$, or $t \in \mathcal{D}_{\mathsf{sched}}(\mathcal{D}_{\mathsf{conf}}(\mathcal{D}_{\mathsf{trans}}(Q)))$ with annotations.*

The following function δ_σ, as sketched in the beginning of the section, extracts a pre-configuration from an MDP $\mathcal{M}$ and a scheduler σ, exposing the three sources of probabilities.

▶ **Definition 6** (δ_σ). *Let $\mathcal{M} = (Q, \mathsf{Act}, \delta)$ be an MDP and σ be a scheduler. We define a function $\delta_\sigma \colon \mathcal{D}(Q)^+ \to \mathcal{D}(\mathcal{D}(\mathcal{D}(Q)))$ as follows: for any $d_0 d_1 \dots d_k \in \mathcal{D}(Q)^+$,*

$$\delta_\sigma(d_0 d_1 \dots d_k) = \sum_{a \in \mathsf{Act}} \sigma(d_0 d_1 ... d_k)(a) \Big| \sum_{q \in Q} d_k(q) \,\Big|\, \delta(q,a) \,\Big\rangle\Big\rangle. \tag{2}$$

The annotations in both (1) and Def. 5 can now be explained using (2).

- The innermost distribution $\delta(q,a)$ on the right-hand side of (2) comes from the transition function $\delta \colon Q \times \mathsf{Act} \to \mathcal{D}(Q)$ of an MDP; so we write $\delta(q,a) \in \mathcal{D}_{\mathsf{trans}}(Q)$.
- The distribution $\Big| \sum_{q \in Q} d_k(q) \,\Big|\, \delta(q,a) \,\Big\rangle\Big\rangle$, that is one-level higher, has probabilities $d_k(q)$ that come from a "configuration" d_k (this terminology is justified by the memoryless case of Def. 9 later). Thus we say it belongs to $\mathcal{D}_{\mathsf{conf}}(\mathcal{D}_{\mathsf{trans}}(Q))$.
- Finally, the whole right-hand side of each of (2) is a probability distribution over elements of $\mathcal{D}_{\mathsf{conf}}(\mathcal{D}_{\mathsf{trans}}(Q))$, where the probabilities (such as $\sigma(d)(a)$) come from the scheduler σ. Therefore, the right-hand side is an element of $\mathcal{D}_{\mathsf{sched}}(\mathcal{D}_{\mathsf{conf}}(\mathcal{D}_{\mathsf{trans}}(Q)))$.

The following defines CM classifiers, whose intuitions have been discussed earlier in Sec. 3.1.

▶ **Definition 7** (chance-mass (CM) classifier). *A* chance-mass (CM) classifier *is a family $\mathbb{X} = (\mathbb{X}_Q)_Q$ of functions indexed by set Q; its Q-component has type $\mathbb{X}_Q : \mathcal{D}(\mathcal{D}(\mathcal{D}(Q))) \to \mathcal{D}(\mathcal{D}(Q))$. When the index Q is obvious we drop it, writing $\mathbb{X} \colon \mathcal{D}(\mathcal{D}(\mathcal{D}(Q))) \to \mathcal{D}(\mathcal{D}(Q))$.*

▶ Remark 8. For our theoretical development, in Def. 7, it is enough to fix an MDP $\mathcal{M} = (Q, \mathsf{Act}, \delta)$ and only consider the Q-component $\mathbb{X}_Q \colon \mathcal{D}(\mathcal{D}(\mathcal{D}(Q))) \to \mathcal{D}(\mathcal{D}(Q))$ of $\mathbb{X}$. We made the above general definition – it is independent of the choice of Q – to emphasize the *uniformity* of $\mathbb{X}$. This uniformity should be formalized as the *naturality in Q*, a central notion from category theory [22]. We will do so in our future work.

We are ready to define a config MC $\mathcal{M}_{\mathbb{X},\sigma}$, a semantic basis on which we formalize e.g. reachability problems. As announced, its transition function is a composition $\mathbb{X} \circ \delta_\sigma$ of 1) a CM classifier $\mathbb{X}$ that chooses the type of semantics (cf. Table 1) and 2) the function δ_σ that exposes three sources of probability in an MDP and forms a pre-configuration.

▶ **Definition 9** (config MC $\mathcal{M}_{\mathbb{X},\sigma}$). *Let $\mathcal{M} = (Q, \mathsf{Act}, \delta)$ be an MDP, $\mathbb{X}$ be a CM classifier and σ be a scheduler. The* configuration Markov chain (config MC) *$\mathcal{M}_{\mathbb{X},\sigma} = \big(\mathcal{D}(Q)^+, \delta_{\mathbb{X},\sigma}\big)$ is a Markov chain carried by the set $\mathcal{D}(Q)^+$. Its transition function $\delta_{\mathbb{X},\sigma} \colon \mathcal{D}(Q)^+ \to \mathcal{D}(\mathcal{D}(Q)^+)$ is defined as follows: for $d_0 d_1 \dots d_k \in \mathcal{D}(Q)^+$,*

$$\delta_{\mathbb{X},\sigma}(d_0 d_1 \dots d_k) = \sum_{d_{k+1} \in \mathcal{D}(Q)} \Big(\mathbb{X}\big(\delta_\sigma(d_0 d_1 \dots d_k)\big)(d_{k+1}) \Big) \, |d_0 d_1 ... d_k d_{k+1}\rangle,$$

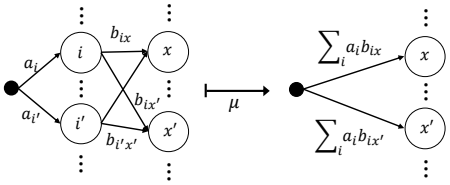


Figure 2 The suppression operator μ.

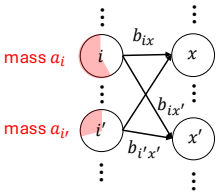


Figure 3 The MC2CM operator λ, its input.

that is, a config MC state $d_0 d_1 \dots d_k$ evolves into $d_0 d_1 \dots d_k d_{k+1}$ with the probability $\big(\mathbb{X}\big(\delta_\sigma(d_0 d_1 \dots d_k)\big)\big)\,(d_{k+1})$. Note that, in this memoryful setting, config MC states record the whole history as sequences and get extended along transitions.

For a memoryless scheduler $\sigma\colon \mathcal{D}(Q) \to \mathcal{D}(\mathsf{Act})$, the config MC $\mathcal{M}_{\mathbb{X},\sigma} = \big(\mathcal{D}(Q), \delta_{\mathbb{X},\sigma}\big)$ is carried by the set $\mathcal{D}(Q)$ of configurations (Def. 3). The function $\delta_{\mathbb{X},\sigma}\colon \mathcal{D}(Q) \to \mathcal{D}(\mathcal{D}(Q))$ is

$$\delta_{\mathbb{X},\sigma}(d) = \sum_{d' \in \mathcal{D}(Q)} \Big(\mathbb{X}\big(\delta_\sigma(d)\big)(d')\Big)\,|d'\rangle, \tag{3}$$

that is, a configuration $d \in \mathcal{D}(Q)$ evolves into $d' \in \mathcal{D}(Q)$ with the probability $\big(\mathbb{X}\big(\delta_\sigma(d)\big)\big)\,(d')$. Note that by the definition of $\mathcal{D}$, the CM classifier $\mathbb{X}\big(\delta_\sigma(d_0 d_1 \dots d_k)\big)$ and $\mathbb{X}\big(\delta_\sigma(d)\big)$ are finitely-supported, hence the use of summation notation here is well-defined.

We formally state the reachability problem. More details are in [28].

▶ **Problem 10** (threshold reachability problem). *Assume the setting of Def. 9. Let $d_0 \in \mathcal{D}(Q)$ and $H \subseteq \mathcal{D}(Q)$, be an* initial configuration *and a* target set*, respectively. We let $\mathsf{Reach}_{\mathbb{X},\sigma}(d_0, H)$ denote the set of paths that start from d_0 and eventually reach H. The* threshold reachability problem *for ξ asks if there exists a scheduler σ such that $\mathbb{P}_{\mathbb{X},\sigma}\big(\mathsf{Reach}_{\mathbb{X},\sigma}(d_0, H)\big) \geq \xi$.*

4 Instances of the Unified Framework and Four Semantics

Now we instantiate the framework and derive the four semantics of MDPs in Tables 1 and 2. We first introduce the three operators λ, μ, η for constructing the CM classifiers $\mathbb{X}$.

▶ **Definition 11** (MC2CM λ, suppression μ, Dirac η). *Let X be a set.*

- *The* MC2CM operator *$\lambda_X\colon \mathcal{D}(\mathcal{D}(X)) \to \mathcal{D}(\mathcal{D}(X))$ on X is defined as follows:*

$$\lambda_X\Big(\sum_{i\in I} a_i \,\Big|\, \sum_{x\in X} b_{ix}|x\rangle\Big\rangle\Big) = \sum_{f\in X^I} \Big(\prod_{i\in I} b_{i,f(i)}\Big) \,\Big|\, \sum_{i\in I} a_i \,|f(i)\rangle\Big\rangle. \tag{4}$$

- *The* suppression operator *$\mu_X\colon \mathcal{D}(\mathcal{D}(X)) \to \mathcal{D}(X)$ on X is defined as follows:*

$$\mu_X\Big(\sum_{i\in I} a_i \,\Big|\, \sum_{x\in X} b_{ix}|x\rangle\Big\rangle\Big) = \sum_{x\in X} \Big(\sum_{i\in I} a_i b_{ix}\Big)|x\rangle. \tag{5}$$

- *The* Dirac operator *$\eta_X\colon X \to \mathcal{D}(X)$ is defined by $\eta_X(x) = 1\,|x\rangle$, that is, $\eta_X(x)(x') = 1$ if $x = x'$ and 0 otherwise.*

Much like in Def. 7, we often drop the subscript X and simply write λ, μ, η.

Here are some intuitions (see [28] for more). The suppression operator μ, as in Fig. 2, suppresses two successive probabilistic branching into one: *tossing coins altogether, instead of one-by-one*. The Dirac operator η turns an element x into a trivial distribution.

The MC2CM operator λ is much less known. Its type with annotations is $\lambda\colon \mathcal{D}_{\mathsf{mass}}(\mathcal{D}_{\mathsf{chan}}(X)) \to \mathcal{D}_{\mathsf{chan}}(\mathcal{D}_{\mathsf{mass}}(X))$; hence the name ("mass-chance to chance-mass"). See Fig. 3. This is how we interpret the input $\sum_{i\in I} a_i \left|\sum_{x\in X} b_{ix}|x\rangle\right\rangle$ of λ; more specifically, we are looking at the mass a_i at each $i \in I$, and each $i \in I$ has the transition probability b_{ix} to each $x \in X$.

Here is the intention of λ: the destination of the mass a_i at each i is chosen by chance (coin toss), and the whole mass a_i goes to the chosen destination. This happens at each $i \in I$, and as a result, each $x \in X$ will acquire the sum of the masses whose destination is x.

This whole process can be modeled as follows.

- Using functions $f\colon I \to X$, we can enumerate all combinations of the destinations $f(i)$ of the mass at each $i \in I$. That is, f is the event "the destination of i is chosen as $f(i)$."
- For each such event $f \in X^I$, the probability for f is $\prod_{i\in I} b_{i,f(i)}$, where $b_{i,f(i)}$ is the probability of $f(i)$ being chosen as the destination of i.
- Once an event f is chosen, the mass at each $i \in I$ is moved accordingly. The resulting mass at $x \in X$ is $\sum_{i\in I \text{ s.t. } f(i)=x} a_i$, and the induced distribution of masses is conveniently represented in the ket notation by $\sum_{i\in I} a_i|f(i)\rangle$.

This explains the right-hand side of (4): it is the distribution over the mass distributions $\sum_{i\in I} a_i|f(i)\rangle$ (where f varies), and the probability for each f given by $\prod_{i\in I} b_{i,f(i)}$.

4.1 Concrete Instances

Now we present, in the concrete level of elements, the four CM classifiers outlined in Table 2. We also present explicit formulas for the config MCs that are derived using them.

▶ **Definition 12** (CM classifiers, concrete instances). *We define four CM classifiers* $\mathbb{X}^{(\mathbf{CS},\mathbf{CT})}, \mathbb{X}^{(\mathbf{MS},\mathbf{CT})}, \mathbb{X}^{(\mathbf{CS},\mathbf{MT})}, \mathbb{X}^{(\mathbf{MS},\mathbf{MT})} : \mathcal{D}_{\mathsf{sched}}(\mathcal{D}_{\mathsf{conf}}(\mathcal{D}_{\mathsf{trans}}(Q))) \to \mathcal{D}_{\mathsf{chan}}(\mathcal{D}_{\mathsf{mass}}(Q))$*:*

$$\mathbb{X}^{(\mathbf{CS},\mathbf{CT})} = \mu_{\mathcal{D}(Q)} \circ \mathcal{D}\lambda_Q, \quad \textit{that is concretely,}$$

$$\mathbb{X}^{(\mathbf{CS},\mathbf{CT})}\left(\sum_{i\in I} a_i \left|\sum_{j\in J} b_{i,j}\Big|\sum_{q\in Q} p_{i,j,q}|q\rangle\Big\rangle\right\rangle\right) = \sum_{i\in I}\sum_{f\in Q^J}\left(a_i \prod_{j\in J} p_{i,j,f(j)}\right)\Big|\sum_{j\in J} b_{i,j}|f(j)\rangle\Big\rangle;$$

$$\mathbb{X}^{(\mathbf{MS},\mathbf{CT})} = \mathcal{D}\mu_{(Q)} \circ \lambda_{\mathcal{D}(Q)} \circ \mathcal{D}\lambda_Q, \quad \textit{that is concretely,}$$

$$\mathbb{X}^{(\mathbf{MS},\mathbf{CT})}\left(\sum_{i\in I} a_i \left|\sum_{j\in J} b_{i,j}\Big|\sum_{q\in Q} p_{i,j,q}|q\rangle\Big\rangle\right\rangle\right) = \sum_{f\in Q^{I\times J}} \prod_{(i,j)\in I\times J} p_{i,j,f(i,j)}\Big|\sum_{(i,j)\in I\times J} a_i b_{i,j}|f(i,j)\rangle\Big\rangle;$$

$$\mathbb{X}^{(\mathbf{CS},\mathbf{MT})} = \mathcal{D}\mu_{(Q)}, \quad \textit{that is concretely,}$$

$$\mathbb{X}^{(\mathbf{CS},\mathbf{MT})}\left(\sum_{i\in I} a_i \left|\sum_{j\in J} b_{i,j}\Big|\sum_{q\in Q} p_{i,j,q}|q\rangle\Big\rangle\right\rangle\right) = \sum_{i\in I} a_i\Big|\sum_{j\in J}\sum_{q\in Q} b_{i,j}p_{i,j,q}|q\rangle\Big\rangle; \textit{ and}$$

$$\mathbb{X}^{(\mathbf{MS},\mathbf{MT})} = \eta_{\mathcal{D}(Q)} \circ \mathcal{D}\mu_Q \circ \mu_{\mathcal{D}(Q)}, \quad \textit{that is concretely,}$$

$$\mathbb{X}^{(\mathbf{MS},\mathbf{MT})}\left(\sum_{i\in I} a_i \left|\sum_{j\in J} b_{i,j}\Big|\sum_{q\in Q} p_{i,j,q}|q\rangle\Big\rangle\right\rangle\right) = \Big|\sum_{i\in I}\sum_{j\in J}\sum_{q\in Q} a_i b_{i,j}p_{i,j,q}|q\rangle\Big\rangle.$$

Here is the illustration of the definition of the $(\mathbf{CS},\mathbf{CT})$ case (the other cases are similar). As demonstrated in Table 3, the CM classifier $\mathbb{X}^{(\mathbf{CS},\mathbf{CT})}$ first applies $\mathcal{D}\lambda_Q$ to the pre-configuration to rearrange the order of the distribution operators and obtain an element

in $\mathcal{D}_{\mathsf{sched}}(\mathcal{D}_{\mathsf{trans}}(\mathcal{D}_{\mathsf{conf}}(Q)))$. Since both the scheduler and the transition distributions are interpreted in the chance manner, we apply the suppression operator $\mu_{\mathcal{D}(Q)}$ to suppress $\mathcal{D}_{\mathsf{sched}}$ and $\mathcal{D}_{\mathsf{trans}}$ to $\mathcal{D}_{\mathsf{chan}}$ and leave $\mathcal{D}_{\mathsf{conf}}$ to $\mathcal{D}_{\mathsf{mass}}$.

Those concrete CM classifiers in Def. 12 instantiate the general definition of config MC (Def. 9) as follows. To avoid complex super- and subscripts, a config MC $\mathcal{M}_{\mathbb{X}^{(\mathbf{CS},\mathbf{CT})},\sigma}$ shall be denoted by $\mathcal{M}_\sigma^{(\mathbf{CS},\mathbf{CT})}$, and similarly for other semantics. Here we focus on memoryless schedulers for readability (cf. (3)); the general case is presented in [28].

▶ **Definition 13** (config MC under concrete semantics, memoryless). *Let $\mathcal{M} = (Q, \mathsf{Act}, \delta)$ be an MDP, and $\sigma\colon \mathcal{D}(Q) \to \mathcal{D}(\mathsf{Act})$ be a (memoryless) scheduler. The four config MCs induced by $\mathcal{M}$ and σ, under the four CM classifiers in Def. 12, are concretely described as follows.*

- *Those config MCs are denoted by $\mathcal{M}_\sigma^{(\mathbf{CS},\mathbf{CT})}, \mathcal{M}_\sigma^{(\mathbf{MS},\mathbf{CT})}, \mathcal{M}_\sigma^{(\mathbf{CS},\mathbf{MT})}, \mathcal{M}_\sigma^{(\mathbf{MS},\mathbf{MT})}$.*
- *They all have $\mathcal{D}(Q)$ as a state space.*
- *Their transition functions, all of type $\mathcal{D}(Q) \to \mathcal{D}(\mathcal{D}(Q))$, are given as follows.*

$$
\begin{aligned}
\delta_{\mathbb{X}^{(\mathbf{CS},\mathbf{CT})},\sigma}(d)(d') &= \sum_{a\in\mathsf{Act}} \sum_{f\in Q^Q \ s.t.\ \mathcal{D}f(d)=d'} \sigma(d)(a)\cdot\left(\prod_{q\in Q}\delta(q,a)\big(f(q)\big)\right),\\
\delta_{\mathbb{X}^{(\mathbf{MS},\mathbf{CT})},\sigma}(d)(d') &= \sum_{f\in Q^{Q\times\mathsf{Act}} \ s.t.\ d'=\mathcal{D}f(d\otimes\sigma(d))} \ \prod_{(q,a)\in Q\times\mathsf{Act}} \delta(q,a)(f(q,a)),\\
\delta_{\mathbb{X}^{(\mathbf{CS},\mathbf{MT})},\sigma}(d)(d') &= \sum_{a\in\mathsf{Act} \ s.t.\ d'=\mu_Q(\mathcal{D}(\delta^\wedge(a))(d))} \sigma(d)(a), \qquad (6)\\
\delta_{\mathbb{X}^{(\mathbf{MS},\mathbf{MT})},\sigma}(d)(d') &= \begin{cases} 1 & \text{if } d' = \sum_{a\in\mathsf{Act}}\sum_{q'\in Q}\sum_{q\in Q}\sigma(d)(a)d(q)\delta(q,a)(q')|q'\rangle,\\ 0 & \text{otherwise.}\end{cases}
\end{aligned}
$$

In the second line, $d\otimes\sigma(d)\in\mathcal{D}(Q\times\mathsf{Act})$ denotes the product, which is a coupling of the configuration $d\in\mathcal{D}(Q)$ and the distribution $\sigma(d)\in\mathcal{D}(\mathsf{Act})$ over actions, that is $(d\otimes\sigma(d))(q,a) = d(q)\cdot\sigma(d)(a)$.

▶ Remark 14. As we discussed in Sec. 1, the $(\mathbf{CS},\mathbf{CT})$ semantics should coincide with the conventional semantics of MDPs, but Def. 13 may not look that way. We can show that 1) the transition function $\delta_{\mathbb{X}^{(\mathbf{CS},\mathbf{CT})},\sigma}$ *preserves Dirac-ness* (i.e. if d is Dirac, then $\delta_{\mathbb{X}^{(\mathbf{CS},\mathbf{CT})},\sigma}(d)(d') > 0$ implies d' is Dirac as well), and 2) the definition in Def. 13 indeed is the usual semantics when d and d' are Dirac (meaning that these configurations are single states with mass 1). A proof is in [28].

▶ Remark 15 (combinations other than the four). We have presented four instantiations of the CM classifier framework, corresponding Table 2. Combinatorially, there are $2^3 = 8$ possible ways of assigning $\mathcal{D}_{\mathsf{sched}}$, $\mathcal{D}_{\mathsf{conf}}$ and $\mathcal{D}_{\mathsf{trans}}$ to $\mathcal{D}_{\mathsf{chan}}$ and $\mathcal{D}_{\mathsf{mass}}$, but we have only four since we always assign $\mathcal{D}_{\mathsf{conf}}$ to $\mathcal{D}_{\mathsf{mass}}$. The other options (where $\mathcal{D}_{\mathsf{conf}}$ is assigned to $\mathcal{D}_{\mathsf{chan}}$) are not considered as they do not conceptually fit the general framework in Sec. 3, where a "state" of a config MC induced by an MDP should be a distribution over the state space of the MDP.

5 Algorithms and Complexity

Now we present several complexity results and two algorithms for Problem 10, with $\mathbb{X}^{(\mathbf{CS},\mathbf{MT})}$ and $\mathbb{X}^{(\mathbf{MS},\mathbf{CT})}$ as CM classifiers (the other two have been studied before). Due to space constraints, all proofs are in Appendix of [28]. For completeness, we start by recalling known results when $\mathbb{X}^{(\mathbf{CS},\mathbf{CT})}$ and $\mathbb{X}^{(\mathbf{MS},\mathbf{MT})}$ are provided as the input for the CM classifier. In this section, we refer to Problem 10 as the *threshold* $\mathbf{S}$*-reachability problem* with the CM classifier provided as $\mathbb{X}^{\mathbf{S}}$, where $\mathbf{S}$ can be $(\mathbf{CS},\mathbf{CT})$, $(\mathbf{MS},\mathbf{CT})$, $(\mathbf{CS},\mathbf{MT})$ or $(\mathbf{MS},\mathbf{MT})$.

Known results on the $(\mathbf{CS}, \mathbf{CT})$ and the $(\mathbf{MS}, \mathbf{MT})$ semantics. The $(\mathbf{CS}, \mathbf{CT})$ semantics models the conventional MDP semantics (Rem. 14). This implies that the threshold $(\mathbf{CS}, \mathbf{CT})$-reachability problem is in Ptime when the initial configuration d_0 is Dirac. The proof is a simple adaption of those for the conventional semantics and can be found in [9].

The $(\mathbf{MS}, \mathbf{MT})$ semantics is about distribution transformers in [1, 2, 4–6, 10, 11, 19, 21]. There, the threshold reachability is shown to be positivity-hard (see [3]), implying that the threshold $(\mathbf{MS}, \mathbf{MT})$-reachability is positivity-hard. (The result in [3] uses local actions but it is easily translated to our formalization with global actions. See Rem. 4.)

5.1 Complexity and Algorithm for the $(\mathrm{MS}, \mathrm{CT})$ Semantics

We first a lower bound for the threshold $(\mathbf{MS}, \mathbf{CT})$-reachability. It is proven by a reduction from the counting variant of the subset sum problem, which is known to be ♯P-complete [25].

▶ **Theorem 16.** *The threshold $(\mathbf{MS}, \mathbf{CT})$-reachability problem is* ♯P-hard.

We show, however, that given an explicit scheduler, the problem becomes positivity-hard. Unlike in conventional $(\mathbf{CS}, \mathbf{CT})$ or distributional $(\mathbf{MS}, \mathbf{MT})$ semantics, here fixing a scheduler does not ease the problem. Thm. 18 is proved by reduction from the reachability problem for Markov chains under the $(\mathbf{MS}, \mathbf{MT})$ semantics (shown positivity-hard in [3]).

▶ **Problem 17** (threshold $(\mathbf{MS}, \mathbf{CT})$-reachability with explicit scheduler)**.** *Given an MDP $\mathcal{M} = (Q, \mathsf{Act}, \delta)$, an initial configuration $d_0 \in \mathcal{D}(Q)$, a scheduler σ, a set of configuration $H \subseteq \mathcal{D}(Q)$ and a threshold $\xi \in [0, 1]$, decide if $\mathbb{P}_{\mathbb{X}(\mathbf{MS},\mathbf{CT}),\sigma}(\mathsf{Reach}_{\mathbb{X}(\mathbf{MS},\mathbf{CT}),\sigma}(d_0, H)) \geq \xi$.*

▶ **Theorem 18** (positivity-hardness when scheduler given)**.** *Problem 17 is Positivity-hard.*

▶ Remark 19. The proof of the above results are presented in [28]. Both of the proof does not require the scheduler to be memoryful, hence the hardness results holds even when the problem is restricted to memoryless schedulers.

A Template-based Synthesis Algorithm for Finitely-Generated Monotone Set

Now we present a template-based synthesis algorithm for solving the threshold $(\mathbf{MS}, \mathbf{CT})$-reachability problem, under the assumption that the target set is provided as a finitely-generated monotone set.

We saw that the threshold reachability problem is hard (Thms 16 and 18) – this is because MDPs under the $(\mathbf{MS}, \mathbf{CT})$ semantics can branch into exponentially many configurations which are expensive to track. Therefore, instead of tracking reachable config's, we directly search for a reachability certificate by template-based synthesis and constraint solving.

Our result is an extension of the template-based algorithm in [5] with *γ-scaled submartingales* introduced in [27]. We also make several changes on the assumption of the target set, namely using upward- or downward-closed sets, which involves some technical novelties. We present a version with upward-closed sets; dealing with downward-closed sets is similar.

We start by defining the γ-scaled configuration submartingale, adapted from [27].

▶ **Definition 20** (γ-scaled configuration submartingale)**.** *Given an MDP $\mathcal{M}$, a memoryless scheduler σ, a target set H and a real number $\gamma \in (0, 1)$, a γ-scaled configuration submartingale is a function $R : \mathcal{D}(Q) \to [0, 1]$ such that the following holds: for each $d \in \mathcal{D}(Q) \setminus H$,*

$$R(d) \;\leq\; \gamma \cdot \sum_{d' \in \mathcal{D}(Q)} R(d') \cdot \delta_{\mathbb{X}(\mathbf{MS},\mathbf{CT}),\sigma}(d)(d'). \tag{7}$$

Note that, in (7), the term $\delta_{\mathbb{X}^{(\mathbf{MS},\mathbf{CT})},\sigma}(d)$ is of type $\mathcal{D}(\mathcal{D}(Q))$ which gives a distribution over successor configurations d' in the config MC under the scheduler σ. Hence, evaluating it with d' gives the probability of reaching d' from d. The sum in (7) can be understood as the expected value of the submartingale R after one-step under the scheduler σ. Note that the sum is well-defined since all distributions in this paper are countably supported.

The next theorem is proved much like in [27].

▶ **Theorem 21** (γ-scaled submartingale witness). *Assume that the input of Problem 10 is given, with $\mathbb{X}^{(\mathbf{MS},\mathbf{CT})}$ as the CM classifier. Let R be a γ-scaled submartingale R for some $\gamma \in (0,1)$. Then $R(d_0) \geq \xi$ implies there exists a memoryless scheduler σ such that $\mathbb{P}(\mathsf{Reach}_{\mathbb{X}^{(\mathbf{MS},\mathbf{CT})},\sigma}(d_0, H)) \geq \xi$. If $R(d_0) \geq \xi$ holds, we say that R is a witness of reachability.*

As usual in template-based synthesis, our algorithm collects constraints – ours are given as a system of inequalities over reals – and solves them. The construction is three steps.

- In the first step, templates for a scheduler σ and a γ-scaled configuration submartingale R are constructed.
- In the second step, constraints are constructed to ensure that a feasible solution of the constraints corresponds to a valid scheduler and a valid γ-scaled submartingale that is a witness of reachability.
- In the final step, the whole constraints are passed to an off-the-shelf polynomial inequality solver to compute a feasible solution (we use Z3). If a feasible solution is found, the algorithm concludes the target set is reachable and return the submartingale as a witness.

Below we briefly describe each step.

Step 1: Setup of Template. The first step is to set up templates for a scheduler and a γ-scaled submartingale. This step is standard and done similarly to other template-based synthesis algorithms; thus we only outline the templates here. For details, see [5] and the references therein.

- **Scheduler**: two sets of template variables $\{\theta_q^a \mid a \in \mathsf{Act}, q \in Q \cup \{0\}\}$ and $\{s_q \mid q \in Q \cup \{0\}\}$ are fixed, such that the target scheduler σ is as follows: for each configuration $d \in \mathcal{D}(Q)$ and action $a \in \mathsf{Act}$, $\sigma(d)(a) = \frac{\theta_0^a + \sum_{q\in Q} \theta_q^a \cdot d(q)}{s_0 + \sum_{q\in Q} s_q \cdot d(q)}$.
- **Submartingale**: a set of template variables $\{r_q \mid q \in Q \cup \{0\}\}$ is fixed so that the target submartingale $R\colon \mathcal{D}(Q) \to [0,1]$ is $R(d) = r_0 + \sum_{q\in Q} r_q \cdot d(q)$.

Step 2: Constraint Collection. We now describe how the constraints for the templates are constructed to enforce the validity and witness property of the templates.

Constraints for Schedulers. To ensure that for any configuration $d \in \mathcal{D}(Q)$, $\sigma(d)$ is a distribution over actions, we require it to satisfy 1) for any action $a \in \mathsf{Act}$, $\sigma(d)(a) \geq 0$ and 2) $\sum_{a\in\mathsf{Act}} \sigma(d)(a) = 1$. Concretely, using the template variables,

$$\Phi_{\mathsf{Schedule}} : \begin{cases} d \in \mathcal{D}(Q) & \implies \bigwedge_{a\in\mathsf{Act}} \left[\theta_0^a + \sum_{q\in Q} \theta_q^a \cdot d(q) \geq 0\right] \\ d \in \mathcal{D}(Q) & \implies s_0 + \sum_{q\in Q} s_q \cdot d(q) \geq 1 \\ d \in \mathcal{D}(Q) & \implies \sum_{a\in\mathsf{Act}} \theta_0^a + \sum_{q\in Q} \theta_q^a \cdot d(q) = s_0 + \sum_{q\in Q} s_q \cdot d(q). \end{cases} \tag{8}$$

Constraints for Submartingales. For submartingales, we have two constraints: 1) for any configuration $d \in \mathcal{D}(Q)$, $R(d) \in [0,1]$ and 2) for any configuration $d \in \mathcal{D}(Q) \setminus H$, $R(d)$ satisfies the submartingale condition in (7). Concretely, using the template variables,

$$\Phi_{\mathsf{Bound}}: \quad d \in \mathcal{D}(Q) \implies 0 \leq r_0 + \textstyle\sum_{q \in Q} r_q \cdot d(q) \leq 1, \tag{9}$$

$$\Phi_{\mathsf{Inductive}}: \quad d \in \mathcal{D}(Q) \setminus H \implies r_0 + \textstyle\sum_{q \in Q} r_q \cdot d(q) \leq \gamma \sum_{a \in \mathsf{Act}} \sigma(q)(a) R(M_a^T d). \tag{10}$$

Constraints for Reachability Certificates. We also impose a constraint so that R is a witness of reachability, i.e. $R(d_0) \geq \xi$. Concretely,

$$\Phi_{\mathsf{Reachable}}: \quad R(d_0) = r_0 + \textstyle\sum_{q \in Q} r_q d_0(q) \geq \xi. \tag{11}$$

Step 2.5: Quantifier Elimination. It is worth noting that all the constraints we obtained in the system are either a simple inequality or a quantified formula in the form of $\forall \mathbf{x} \in \mathbb{R}^n$, $\Phi(x) \geq 0 \implies \psi(x) \geq 0$, where Φ is a linear function and ψ is a polynomial. Therefore, the well-known Farkas's lemma [12] and Handelman's theorem [16] can be applied to perform quantifier elimination, which reduce the problem to solving a system of polynomial inequalities over reals. Further details are presented in [28].

Step 3: Constraint Solving. Once all constraints are translated to an existential problem over reals, the system is then passed to a polynomial inequality solver for finding a feasible solution. If a feasible solution is found, then we claim that the target set H is reachable.

▶ **Theorem 22.** *The algorithm is sound, with a runtime in* 2EXPTIME.

The soundness proof follows directly from Thm. 21. For the time complexity, it takes exponential time (in the number of generators of H) to construct all the constraints, and since existential theory of reals is in PSPACE, hence we obtain 2EXPTIME in total.

5.2 Complexity and Algorithm for the $(\mathbf{CS}, \mathbf{MT})$ Semantics

Complexity Results. In the $(\mathbf{CS}, \mathbf{MT})$ semantics, similarly to the conventional $(\mathbf{CS}, \mathbf{CT})$ semantics, one can replace a mixed scheduler with a pure scheduler without decreasing the reachability probability. Intuitively, this is achieved by inductively choosing the action with the largest probability of reaching the target set.

▶ **Lemma 23.** *Assume the setting of Problem 10. If there is a mixed scheduler* $\sigma\colon \mathcal{D}(Q)^+ \to \mathcal{D}(\mathsf{Act})$ *such that* $\mathbb{P}(\mathsf{Reach}_{\mathbb{X}(\mathbf{CS},\mathbf{MT}),\sigma}(d_0, H)) \geq \xi$*, then there is a pure scheduler* $\sigma'\colon \mathcal{D}(Q)^+ \to \mathsf{Act}$ *(interpreted as a scheduler that only consists of Dirac distributions over* Act *in our framework), such that* $\mathbb{P}(\mathsf{Reach}_{\mathbb{X}(\mathbf{CS},\mathbf{MT}),\sigma}(d_0, H)) \geq \xi$.

Therefore we can limit our search to pure (instead of mixed) schedulers. Furthermore we notice the following: in the concrete description of (6), if σ is a pure scheduler, the probability amplitude $\sigma(d)(a)$ is either 0 or 1. All these reduce, in the case of the $(\mathbf{CS}, \mathbf{MT})$ semantics, the original problem (Problem 10) to the following *deterministic* problem.

▶ **Problem 24** (deterministic $(\mathbf{CS}, \mathbf{MT})$-reachability)**.** *In the setting of Problem 10, for a finite sequence* $\bar{a} = a_1 a_2 ... a_n \in \mathsf{Act}^*$ *of actions, let* $d_{\bar{a},1}, ... d_{\bar{a},n} \in \mathcal{D}(Q)^*$ *be a sequence of configurations and* $\sigma_{\bar{a}}$ *be a pure scheduler such that for any* $j \in [n]$*, the following is satisfied:* $\sigma_{\bar{(a)}}(d_0 d_{\bar{a},1} ... d_{\bar{a},j-1})(a_j) = 1$ *and* $\delta_{\mathbb{X}(\mathbf{CS},\mathbf{MT}),\sigma_{\bar{a}}}(d_0 d_{\bar{a},1} ... d_{\bar{a},j-1})(d_{\bar{a},j}) = 1$*. Note that the scheduler is potentially memoryful.*

The deterministic $(\mathbf{CS}, \mathbf{MT})$-reachability problem *asks if there exists a finite sequence* $\bar{a} = a_1 a_2 ... a_n \in \mathsf{Act}^*$ *of actions and a natural number* $n \in \mathbb{N}$ *such that* $d_{\bar{a},n} \in H$.

Now the following result is proved by reduction from the emptiness problem of probabilistic finite automaton (PFA). The latter is undecidable; see e.g. [14]. Further, we also show that the problem remains undecidable even when the target set H is finitely generated and monotone. Both results only hold for searching a memoryful scheduler.

▶ **Theorem 25** (undecidability of $(\mathbf{CS}, \mathbf{MT})$-reachability)**.** *The threshold $(\mathbf{CS}, \mathbf{MT})$-reachability is undecidable even when the target set H is restricted to a finitely-generated monotone set.*

An Antichain-based Algorithm for Finitely-generated Monotone Set

Despite the complexity, here we attempt to provide an algorithm for solving Problem 24 with a target set H being finitely generated and upward-closed (similarly to Sec. 5.1, the adaption to downward-closed sets is straightforward). We motivate the use of antichain – used e.g. in [13] – by a simple observation that all stochastic matrices are monotone with respect to the element-wise order. That is, let $M \in \mathsf{Stoc}(n)$ be a stochastic matrix, then for any $\mathbf{x}, \mathbf{y} \in [0,1]^n$, $\mathbf{x} \preceq \mathbf{y}$ implies $M^T\mathbf{x} \preceq M^T\mathbf{y}$. We come to the following simple lemma.

▶ **Lemma 26** (pullback of upward-closed set)**.** *Let H be an upward-closed set and $M \in \mathsf{Stoc}(n)$ be a stochastic matrix. Then the pullback of H with respect to M, denoted as $M^\sharp H$ and defined as $M^\sharp H = \{\mathbf{x} \in [0,1]^n \mid M^T\mathbf{x} \in H\}$, is upward closed.*

Hence inductively, the set of all initial configurations that are reachable to H is also upward-closed, which motivates the usage of antichains since they are a canonical representation of upward-closed sets. Formally, we connect this observation to Problem 24 with the following theorem, which characterizes the set of configurations reaching H as the least fixed point of an operator that preserves upward-closeness.

▶ **Theorem 27** (reachable configuration as a least fixed point)**.** *Given an MDP $\mathcal{M} = (Q, \mathsf{Act}, \mathsf{Mat})$, an initial configuration $d_0 \in \mathcal{D}(Q)$ and a finitely-generated upward-closed set H, let $B : 2^{[0,1]^Q} \to 2^{[0,1]^Q}$ be a function defined by $B(S) = H \cup \bigcup_{M \in \mathsf{Mat}} M^\sharp S$. Then the sequence $\left(B^j(\bot)\right)_{j \in \mathbb{N}}$ converges to its least fixed point μB and H is reachable from d_0 if and only if $d_0 \in \mu B$.*

The $(\mathbf{CS}, \mathbf{MT})$-reachability problem is undecidable in general (Thm. 25); thus computing the least fixed point of B is not always possible. Nevertheless, we can approximate it and provide a sound but necessarily incomplete algorithm. Since $\bot$ is by definition upward closed, it follows that all approximants $B^k(\bot)$ are upward closed. Hence algorithmically, it suffices for us to keep track of the minimal elements of each $B^k(\bot)$, which can be maintained efficiently through the antichain data structure.

The only challenge that remains is to compute the bottom (the subset of $\preceq$-minimal elements) of an upward-closed set. Specifically, in the current setting, we need to compute $\lfloor M^\sharp S \rfloor = \lfloor \{\mathbf{y} \in [0,1]^{|Q|} \mid M^T\mathbf{y} \succeq \mathbf{x} \text{ for some } \mathbf{x} \in S\} \rfloor$ for $M \in \mathsf{Mat}$. Since we assume an antichain structure is always maintained in the algorithm, this problem can be reduced to computing the set $\lfloor \{\mathbf{y} \in [0,1]^{|Q|} \mid M^T\mathbf{y} \succeq \mathbf{x}\} \rfloor$ for each $\mathbf{x} \in \lfloor S \rfloor$ and merging them.

Our algorithm solves the last problem (for each $\mathbf{x} \in \lfloor S \rfloor$) by iteratively solving the following sequence of minimization problems, increasing k over time: we randomly sample $(\mathbf{q}_j)_{j \in [k-1]} \in [\,|Q|\,]^{k-1}$ (recall the notation $[n]$ from Sec. 2), and solve

$$\mathbf{y}_k^* = \operatorname*{argmin}_{\mathbf{y}} \mathbf{1}^T\mathbf{y} \quad \text{subject to} \quad M^T\mathbf{y} \succeq \mathbf{x}, \quad \mathbf{1}^T\mathbf{y} \leq 1, \quad \mathbf{y} \succeq \mathbf{0}, \quad \text{and} \quad \forall j < k,\ (\mathbf{y})_{\mathbf{q}_j} < (\mathbf{y}_j^*)_{\mathbf{q}_j}. \tag{12}$$

The intuition is as follows. Our goal is to iteratively obtain elements $\mathbf{y}_1^*, \mathbf{y}_2^*, \ldots$ of the set $S = \lfloor \{\mathbf{y} \in [0,1]^{|Q|} \mid M^T\mathbf{y} \succeq \mathbf{x}\} \rfloor$. For the first element $\mathbf{y}_1^*$, the last constraint in (12) is vacuous, and we obtain an element of S that is minimal wrt. $\preceq$ (here it is important that $\mathbf{y} \preceq \mathbf{y}'$ implies $\mathbf{1}^T\mathbf{y} \leq \mathbf{1}^T\mathbf{y}'$). For the step case with $k > 1$, the last constraint in (12) enforces that $\mathbf{y}_j^* \not\preceq \mathbf{y}_k^*$ for each $j \in [k-1]$, where $\mathbf{q}_j \in |Q|$ is the "guessed" index that witnesses $\mathbf{y}_j^* \not\preceq \mathbf{y}_k^*$. That $\mathbf{y}_k^* \preceq \mathbf{y}_j^*$ is prohibited, too, for each $j < k$, since otherwise $\mathbf{y}_k^*$ must have been obtained in the j-th round instead of $\mathbf{y}_j^*$.

This problem is an instance of linear programming which can be solved by any off-the-shelf LP solver in PTime. If the solver does not return a solution, another $\mathbf{q}$ will be sampled and used to solve for $\mathbf{y}_k^*$. The process terminates when either K solutions $\mathbf{y}_1^*, \ldots, \mathbf{y}_K^*$ are found, or the solver fails to find $\mathbf{y}_k^*$ for some k after L different samples of $(\mathbf{q}_j)_j$'s (K and L are hyperparameters). The set of solutions $\{\mathbf{y}_k^* \mid k \in [K]\}$ is then used to approximate the set $\lfloor \{\mathbf{y} \in [0,1]^{|Q|} \mid M^T\mathbf{y} \succeq \mathbf{x}\} \rfloor$.

▶ **Lemma 28.** *Given a sub-distribution $\mathbf{x} \in [0,1]^n$, a stochastic matrix $M \in \mathsf{Stoc}(n)$, and parameters $K, L \in \mathbb{N}$, let $\mathcal{Y}_{K,L}(\mathbf{x})$ denotes the set of solutions obtained by the above described procedure labelled by parameters K and L, then $\mathcal{Y}_{K,L}(\mathbf{x}) \subseteq \lfloor \{\mathbf{y} \in [0,1]^{|Q|} \mid M^T\mathbf{y} \succeq \mathbf{x}\} \rfloor$. Moreover, if the set $\lfloor \{\mathbf{y} \in [0,1]^{|Q|} \mid M\mathbf{y} \succeq \mathbf{x}\} \rfloor$ is finitely generated, then we have $\mathcal{Y}_{K,L}(\mathbf{x}) = \lfloor \{\mathbf{y} \in [0,1]^{|Q|} \mid M\mathbf{y} \succeq \mathbf{x}\} \rfloor$ almost surely for large enough K and L.*

Algorithm 1 Backward projection algorithm.

Input: an MDP $\mathcal{M} = (Q, \mathsf{Act}, \mathsf{Mat})$, an initial configuration $d_0 \in \mathcal{D}(Q)$, a target set antichain $\lfloor H \rfloor \subset [0,1]^{|Q|}$, parameters $K, L \in \mathbb{N}$
Output: True if d_0 is reachable to H

```
S ← {}
S' ← ⌊H⌋
do
    S ← S'
    S' ← ⌊H⌋
    for M ∈ Mat do
        for x ∈ S do
            {y*_k | k ∈ [K]} ← Solve the problem in (12) against M,x with parameters K, L
            S' ← ⌊S' ∪ {y*_k | k ∈ [K]}⌋
    if d_0 ∈ S' then return True
while S ≠ S'
if d_0 ∈ S' then return True
```

Finally, we formalize the algorithm and its correctness. The algorithm takes as input an MDP $\mathcal{M} = (Q, \mathsf{Act}, \mathsf{Mat})$, an initial configuration $d_0 \in \mathcal{D}(Q)$, a target set H provided as $\lfloor H \rfloor$ and two parameters $K, L \in \mathbb{N}$, and returns True if d_0 is reachable to H. It starts with a set $S = \lfloor H \rfloor$ and iteratively updates S by computing $\lfloor B(S) \rfloor$ through solving the optimization problem described above. The algorithm only terminates when it witnesses a reachability or the set of configurations reachable to H stabilizes, that is, $d_0 \in S$ or $S = S'$. The full pseudocode is provided in Algorithm 1. The termination is necessarily not guaranteed since the problem is undecidable (Thm. 25). But from Lem. 28 we can conclude that the algorithm always computes an under-approximation of the set of configurations reaching H, hence as a direct consequence of Thm. 27, we have the following theorem.

▶ **Theorem 29.** *Algorithm 1 is a sound algorithm for solving the $(\mathbf{CS}, \mathbf{MT})$-reachability problem with finitely-generated monotone target set H.*

6 Prototype Implementation and Experiments

While the primary focus of this paper is the theoretical development of the unifying framework and algorithms in the two new semantics, we also implemented prototypes of the algorithms in Sections 5.1 and 5.2, as a proof of concept. Details of the experiment setup and benchmarks are presented in [28]. We tested the implementations against various models from [4, 10] and also our motivating examples *Casino* and *Exam* from Sec. 1.

Despite the hardness of the problem, the antichain algorithm in Sec. 5.2 on average solves a problem with three to five states within a few seconds. Most runs terminate before reaching a loop limit and prove the reachability. The template-based algorithm in Sec. 5.1 on average solves a problem with 3 to 5 states within the range of a few seconds to a hundred seconds. We were able to find a certificate that witnesses reachability in all our examples by considering only polynomials up to degree 4. Complete results are provided in [28].In summary, our implementation indicates that the algorithms work on small models, at least, and have the potential to be further developed for real-world applications.

7 Conclusion

In this paper, we presented a unified framework for defining semantics for MDPs that distinguishes randomness arising from schedulers from randomness in transitions. By fixing which get interpreted distributionally (*mass*) and which probabilistically (*chance*), we obtain four semantics of which two are new. We proved hardness results for the reachability problem with monotone target states for the two new semantics and developed sound algorithms using completely different techniques. As future work, it would be interesting to use the framework defined here, e.g., for obtaining the semantics for *quantum* MCs and MDPs that have been recently considered in the verification community [15, 29].

References

1 Rajab Aghamov, Christel Baier, Toghrul Karimov, Joris Nieuwveld, Joël Ouaknine, Jakob Piribauer, and Mihir Vahanwala. Model checking markov chains as distribution transformers. In Nils Jansen, Sebastian Junges, Benjamin Lucien Kaminski, Christoph Matheja, Thomas Noll, Tim Quatmann, Mariëlle Stoelinga, and Matthias Volk, editors, *Principles of Verification: Cycling the Probabilistic Landscape - Essays Dedicated to Joost-Pieter Katoen on the Occasion of His 60th Birthday, Part II*, volume 15261 of *Lecture Notes in Computer Science*, pages 293–313. Springer, 2024. doi:10.1007/978-3-031-75775-4_13.

2 Manindra Agrawal, S. Akshay, Blaise Genest, and P. S. Thiagarajan. Approximate verification of the symbolic dynamics of markov chains. *J. ACM*, 62(1):2:1–2:34, 2015. doi:10.1145/2629417.

3 S. Akshay, Timos Antonopoulos, Joël Ouaknine, and James Worrell. Reachability problems for markov chains. *Information Processing Letters*, 115(2):155–158, 2015. doi:10.1016/j.ipl.2014.08.013.

4 S. Akshay, Krishnendu Chatterjee, Tobias Meggendorfer, and Đorđe Žikelić. Mdps as distribution transformers: Affine invariant synthesis for safety objectives. In *Computer Aided Verification: 35th International Conference, CAV 2023, Paris, France, July 17–22, 2023, Proceedings, Part III*, pages 86–112, Berlin, Heidelberg, 2023. Springer-Verlag. doi:10.1007/978-3-031-37709-9_5.

5 S. Akshay, Krishnendu Chatterjee, Tobias Meggendorfer, and Đorđe Žikelić. Certified policy verification and synthesis for mdps under distributional reach-avoidance properties. In *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence*, IJCAI '24, 2024. doi:10.24963/ijcai.2024/1.

6 S. Akshay, Blaise Genest, and Nikhil Vyas. Distribution-based objectives for markov decision processes. In *LICS*, pages 36–45. ACM, 2018. doi:10.1145/3209108.3209185.

7 Sanjeev Arora and Boaz Barak. *Computational Complexity - A Modern Approach*. Cambridge University Press, 2009. URL: http://www.cambridge.org/catalogue/catalogue.asp?isbn=9780521424264.

8 Steve Awodey. *Category Theory*. Oxford Logic Guides. Oxford Univ. Press, 2006.

9 Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT press, 2008.

10 Rohit Chadha, Vijay Anand Korthikanti, Mahesh Viswanathan, Gul Agha, and YoungMin Kwon. Model checking mdps with a unique compact invariant set of distributions. In *Eighth International Conference on Quantitative Evaluation of Systems, QEST 2011, Aachen, Germany, 5-8 September, 2011*, pages 121–130. IEEE Computer Society, 2011. doi:10.1109/QEST.2011.22.

11 Laurent Doyen, Thierry Massart, and Mahsa Shirmohammadi. Limit synchronization in markov decision processes. In *FoSSaCS*, volume 8412 of *Lecture Notes in Computer Science*, pages 58–72. Springer, 2014. doi:10.1007/978-3-642-54830-7_4.

12 Julius Farkas. Theorie der einfachen ungleichungen. *Journal für die reine und angewandte Mathematik (Crelles Journal)*, 1902(124):1–27, 1902.

13 Emmanuel Filiot, Naiyong Jin, and Jean-François Raskin. Antichains and compositional algorithms for ltl synthesis. *Form. Methods Syst. Des.*, 39(3):261–296, December 2011. doi:10.1007/s10703-011-0115-3.

14 Hugo Gimbert and Youssouf Oualhadj. Probabilistic automata on finite words: Decidable and undecidable problems. In Samson Abramsky, Cyril Gavoille, Claude Kirchner, Friedhelm Meyer auf der Heide, and Paul G. Spirakis, editors, *Automata, Languages and Programming, 37th International Colloquium, ICALP 2010, Bordeaux, France, July 6-10, 2010, Proceedings, Part II*, volume 6199 of *Lecture Notes in Computer Science*, pages 527–538. Springer, 2010. doi:10.1007/978-3-642-14162-1_44.

15 Ji Guan, Yuan Feng, Andrea Turrini, and Mingsheng Ying. Measurement-based verification of quantum markov chains. In Arie Gurfinkel and Vijay Ganesh, editors, *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part III*, volume 14683 of *Lecture Notes in Computer Science*, pages 533–554. Springer, 2024. doi:10.1007/978-3-031-65633-0_24.

16 David Handelman. Representing polynomials by positive linear functions on compact convex polyhedra. *Pacific Journal of Mathematics*, 132(1):35–62, 1988.

17 Christian Hensel, Sebastian Junges, Joost-Pieter Katoen, Tim Quatmann, and Matthias Volk. The probabilistic model checker storm. *Int. J. Softw. Tools Technol. Transf.*, 24(4):589–610, 2022. doi:10.1007/S10009-021-00633-Z.

18 Bart Jacobs. *Introduction to Coalgebra: Towards Mathematics of States and Observation*, volume 59 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2016. doi:10.1017/CBO9781316823187.

19 Vijay Anand Korthikanti, Mahesh Viswanathan, Gul Agha, and YoungMin Kwon. Reasoning about mdps as transformers of probability distributions. In *QEST*, pages 199–208. IEEE Computer Society, 2010. doi:10.1109/QEST.2010.35.

20 Marta Z. Kwiatkowska, Gethin Norman, and David Parker. The PRISM benchmark suite. In *Ninth International Conference on Quantitative Evaluation of Systems, QEST 2012, London, United Kingdom, September 17-20, 2012*, pages 203–204. IEEE Computer Society, 2012. doi:10.1109/QEST.2012.14.

21 YoungMin Kwon and Gul A. Agha. Verifying the evolution of probability distributions governed by a DTMC. *IEEE Trans. Software Eng.*, 37(1):126–141, 2011. doi:10.1109/TSE.2010.80.

22 S. Mac Lane. *Categories for the Working Mathematician*. Springer, Berlin, 2nd edition, 1998.

23 Joël Ouaknine and James Worrell. Decision problems for linear recurrence sequences. In *RP*, volume 7550 of *Lecture Notes in Computer Science*, pages 21–28. Springer, 2012. doi:10.1007/978-3-642-33512-9_3.

24 Joël Ouaknine and James Worrell. On linear recurrence sequences and loop termination. *ACM SIGLOG News*, 2(2):4–13, 2015. doi:10.1145/2766189.2766191.

25 Christos H. Papadimitriou. *Computational complexity*. Addison-Wesley, 1994.

26 Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley Series in Probability and Statistics. Wiley, 1994. doi:10.1002/9780470316887.

27 Toru Takisaka, Yuichiro Oyabu, Natsuki Urabe, and Ichiro Hasuo. Ranking and repulsing supermartingales for reachability in randomized programs. *ACM Trans. Program. Lang. Syst.*, 43(2), June 2021. doi:10.1145/3450967.

28 Yun Chen Tsai, Kittiphon Phalakarn, S. Akshay, and Ichiro Hasuo. Chance and mass interpretations of probabilities in markov decision processes (extended version). *arXiv preprint*, 2025.

29 Mingsheng Ying, Yuan Feng, and Shenggang Ying. Optimal policies for quantum markov decision processes. *Int. J. Autom. Comput.*, 18(3):410–421, 2021. doi:10.1007/S11633-021-1278-Z.

New Fault Domains for Conformance Testing of Finite State Machines

Frits Vaandrager
Radboud University, Nijmegen, The Netherlands

Ivo Melse
Radboud University, Nijmegen, The Netherlands

Abstract

A fault domain reflects a tester's assumptions about faults that may occur in an implementation and that need to be detected during testing. A fault domain that has been widely studied in the literature on black-box conformance testing is the class of finite state machines (FSMs) with at most m states. Numerous strategies for generating test suites have been proposed that guarantee fault coverage for this class. These so-called m-complete test suites grow exponentially in $m - n$, where n is the number of states of the specification, so one can only run them for small values of $m - n$. But the assumption that $m - n$ is small is not realistic in practice. In his seminal paper from 1964, Hennie raised the challenge to design checking experiments in which the number of states may increase appreciably. In order to solve this long-standing open problem, we propose (much larger) fault domains that capture the assumption that all states in an implementation can be reached by first performing a sequence from some set A (typically a state cover for the specification), followed by k arbitrary inputs, for some small k. The number of states of FSMs in these fault domains grows exponentially in k. We present a sufficient condition for k-A-*completeness* of test suites with respect to these fault domains. Our condition implies k-A-completeness of two prominent m-complete test suite generation strategies, the Wp and HSI methods. Thus these strategies are complete for much larger fault domains than those for which they were originally designed, and thereby solve Hennie's challenge. We show that three other prominent m-complete methods (H, SPY and SPYH) do not always generate k-A-complete test suites.

2012 ACM Subject Classification Theory of computation

Keywords and phrases conformance testing, finite state machines, Mealy machines, apartness, observation tree, fault domains, k-A-complete test suites

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2025.34

Related Version *Full Version*: `https://arxiv.org/abs/2410.19405`

Funding Supported by NWO project OCENW.M.23.155, Evidence-Driven Black-Box Checking (EVI).

Acknowledgements This article grew out of earlier work of the first author [35, 37] and the bachelor thesis of the second author [21]. An early version was made available on arXiv [36]. Many thanks to Kirill Bogdanov, Paul Fiterău-Broştean, Dennis Hendriks, Bálint Kocsis, Bram Pellen, Jurriaan Rot, and the anonymous reviewers for their insightfull feedback on earlier versions.

1 Introduction

We revisit the classic problem of black-box conformance testing [18] in a simple setting in which both specifications and implementations can be described as (deterministic, complete) finite state machines (FSMs), a.k.a. Mealy machines. Ideally, given a specification FSM $\mathcal{S}$, a tester would like to have a finite set of tests T that is complete in the sense that an implementation FSM $\mathcal{M}$ will pass all tests in T if and only if $\mathcal{M}$ is equivalent to S. Unfortunately, such a test suite does not exist: if N is the number of inputs in the longest test in T then an implementation $\mathcal{M}$ may behave like $\mathcal{S}$ for the first N inputs, but differently from that point onwards. Even though $\mathcal{M}$ is not equivalent to $\mathcal{S}$, it will pass all tests in

© Frits Vaandrager and Ivo Melse;
licensed under Creative Commons License CC-BY 4.0

T. This motivates the use of a *fault domain*, a collection of FSMs that reflects the tester's assumptions about faults that may occur in an implementation and that need to be detected during testing. The challenge then becomes to define a fault domain that captures *realistic assumptions* about possible faults, but still allows for the design of *sufficiently small* test suites that are complete for the fault domain and can be run within reasonable time.

A fault domain that has been widely studied is the set $\mathcal{U}_m$ of finite state machines (FSMs) with at most m states. Test suites that detect any fault in this class are called *m-complete*. The idea of m-complete test suites can be traced back to Moore [23] and Hennie [12]. Numerous methods for constructing m-complete test suites have been proposed, for different types of transition system models, see for instance [39, 3, 41, 9, 28, 26, 18, 6, 27, 30, 31, 32, 38, 22, 16, 35, 10]. We refer to [18, 5, 22] for overviews and further references. The interest in m-complete test suites is somewhat surprising, given that in a black-box setting there is typically no sensible way to bound the number of possible states of an implementation to a small m. After all, each additional Boolean variable in an implementation potentially doubles the number of extra states. This is problematic in practice, since the size of m-complete test suites generated by existing methods grows exponentially in $m - n$, where n is the number of states of the specification. Actually, Moore [23] was not aiming for practical methods and only described *gedanken-experiments*. He introduced the example of combination lock machines, and was therefore well aware of the combinatorial explosions in m-complete test suites. In his seminal paper from 1964, Hennie [12] also observed the exponential blow-up in m-complete test suites and wrote "Further work is needed before it will be practical to design checking experiments in which the number of states may increase appreciably." In two classic papers, Vasilevskii [39] and Chow [3] independently showed that m-complete test suites can be constructed with a size that is polynomial in the size of the specification FSM, for a fixed value of $k = m - n$. Nevertheless, the test suites generated by their W-method grow exponentially in k, and so they did not solve the problem raised by Hennie [12].

In this article, we solve the long-standing open problem of Hennie [12] as follows:

1. As an alternative for m-completeness, we propose fault domains $\mathcal{U}_k^A$ that contain all FSMs in which any state can be reached by first performing a sequence from some set A (typically a state cover for the specification), followed by k arbitrary inputs, for some small k. These fault domains contain FSMs with a number of extra states that grows exponentially in k.
2. Based on ideas from [6, 35, 37], we present a sufficient condition for *k-A-completeness* of test suites with respect to these fault domains, phrased entirely in terms of properties of their testing tree. We present a $\Theta(N^2)$-time algorithm to check this condition for a testing tree with N states.
3. We show that our sufficient condition implies k-A-completeness of two prominent approaches for test suite generation: the Wp-method of Fujiwara et al [9], and the HSI-method of Luo et al [19] and Petrenko et al [41, 28]. The W-method of Vasilevskii [39] and Chow [3], and the UIOv-method of Chan et al [2] are instances of the Wp-method, and the ADS-method of Lee & Yannakakis [17] and the hybrid ADS method of Smeenk et al [31] are instances of the HSI-method. This means that k-A-completeness of these methods follows as well. Hence these m-complete test suite generation methods are complete for much larger fault domains than those for which they were designed originally.
4. We present counterexamples showing that three other prominent test generation methods, the H-method of Dorofeeva et al [6], the SPY-method of Simão, Petrenko and Yevtushenko [30] and the SPYH-method of Soucha and Bogdanov [32], do not always generate k-A-complete test suites.

The rest of this article is structured as follows. First, Section 2 recalls some basic definitions regarding (partial) Mealy machines, observation trees, and test suites. Section 3 introduces k-A-complete test suites, and shows how they strengthen the notion of m-completeness. Next, we present our sufficient condition for k-A-completeness in Section 4. Based on this condition (and its proof), Section 5 establishes k-A-completeness of the Wp and HSI methods, and m-completeness of the H-method. Finally, Section 6, discusses implications of our results and directions for future research. All proofs are deferred to the full version of this article available on arXiv.

2 Preliminaries

In this section, we recall a number of key concepts that play a role in this article: partial functions, sequences, Mealy machines, observation trees, and test suites.

2.1 Partial Functions and Sequences

We write $f\colon X \rightharpoonup Y$ to denote that f is a partial function from X to Y and write $f(x)\!\downarrow$ to mean that f is defined on x, that is, $\exists y \in Y\colon f(x) = y$, and conversely write $f(x)\!\uparrow$ if f is undefined for x. Often, we identify a partial function $f\colon X \rightharpoonup Y$ with the set $\{(x,y) \in X \times Y \mid f(x) = y\}$. We use Kleene equality on partial functions, which states that on a given argument either both functions are undefined, or both are defined and their values on that argument are equal.

Throughout this paper, we fix a nonempty, finite set I of *inputs* and a set O of *outputs*. We use standard notations for sequences. If X is a set then X^* denotes the set of finite *sequences* (also called *words*) over X. For k a natural number, $X^{\leq k}$ denotes the set of sequences over X with length at most k. We write ϵ to denote the empty sequence, X^+ for the set $X^* \setminus \{\epsilon\}$, x to denote the sequence consisting of a single element $x \in X$, and $\sigma \cdot \rho$ (or simply $\sigma\rho$) to denote the concatenation of two sequences $\sigma, \rho \in X^*$. The concatenation operation is extended to sets of sequences by pointwise extension. We write $|\sigma|$ to denote the length of sequence σ. For a sequence $\tau = \rho\,\sigma$ we say that ρ and σ are a prefix and a suffix of τ, respectively. We write $\rho \leq \tau$ iff ρ is a prefix of τ. A set $W \subseteq X^*$ is *prefix-closed* if any prefix of a word in W is also in W, that is, for all $\rho, \tau \in X^*$ with $\rho \leq \tau$, $\tau \in W$ implies $\rho \in W$. For $W \subseteq X^*$, $\mathit{Pref}(W)$ denotes the *prefix-closure* of W, that is, the set $\{\rho \in X^* \mid \exists \tau \in W : \rho \leq \tau\}$ of all prefixes of elements of W. If $\sigma = x\rho$ is a word over X with $x \in X$, then we write $\mathsf{hd}(\sigma)$ for x, and $\mathsf{tl}(\sigma)$ for ρ.

2.2 Mealy machines

Next, we recall the definition of Finite State Machines (FSMs) a.k.a. Mealy machines.

▶ **Definition 2.1** (Mealy machine). *A* Mealy machine *is a tuple $\mathcal{M} = (Q, q_0, \delta, \lambda)$, where Q is a finite set of* states*, $q_0 \in Q$ is the* initial state*, $\delta\colon Q \times I \rightharpoonup Q$ is a (partial)* transition function*, and $\lambda\colon Q \times I \rightharpoonup O$ is a (partial)* output function *that satisfies $\lambda(q,i)\!\downarrow \Leftrightarrow \delta(q,i)\!\downarrow$, for $q \in Q$ and $i \in I$. We use superscript $\mathcal{M}$ to disambiguate to which Mealy machine we refer, e.g. $Q^{\mathcal{M}}$, $q_0^{\mathcal{M}}$, $\delta^{\mathcal{M}}$ and $\lambda^{\mathcal{M}}$. We write $q \xrightarrow{i/o} q'$ to denote $\lambda(q,i) = o$ and $\delta(q,i) = q'$. We call a state $q \in Q$* complete *iff q has an outgoing transition for each input, that is, $\delta(q,i)\!\downarrow$, for all $i \in I$. A set of states $W \subseteq Q$ is* complete *iff each state in W is complete. The Mealy machine $\mathcal{M}$ is* complete *iff Q is complete. The transition and output functions are lifted to sequences in the usual way. Let $q, q' \in Q$, $\sigma \in I^*$ and $\rho \in O^*$. We write $q \xrightarrow{\sigma/\rho} q'$ to denote*

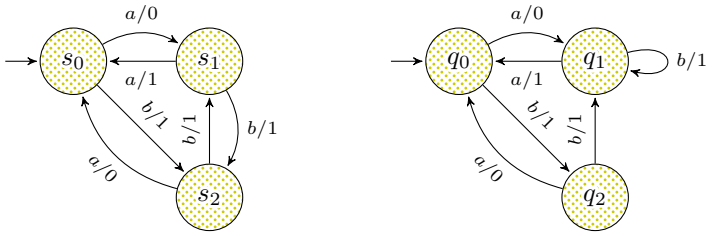


Figure 1 A specification $\mathcal{S}$ (left) and an inequivalent implementation $\mathcal{M}$ (right).

$\lambda(q,\sigma) = \rho$ *and* $\delta(q,\sigma) = q'$. *We write* $q \xrightarrow{\sigma/\rho}$ *if there is a* $q' \in Q$ *with* $q \xrightarrow{\sigma/\rho} q'$, *we write* $q \xrightarrow{\sigma} q'$ *if there is a* $\rho \in O^*$ *with* $q \xrightarrow{\sigma/\rho} q'$, *and we write* $q \xrightarrow{+} q'$ *if there is a* $\sigma \in I^+$ *with* $q \xrightarrow{\sigma} q'$. *If* $q_0 \xrightarrow{\sigma} q$ *then we say that* q *is* reachable *via* σ.

A state cover *for* $\mathcal{M}$ *is a finite, prefix-closed set of input sequences* $A \subset I^*$ *such that, for every* $q \in Q$, *there is a* $\sigma \in A$ *such that* q *is reachable via* σ. *A state cover* A *is* minimal *if each state of* $\mathcal{M}$ *is reached by exactly one sequence from* A. $\mathcal{M}$ *is* initially connected *if it has a state cover. We will only consider Mealy machines that are initially connected.*

▶ **Definition 2.2** (Semantics and minimality). *The* semantics *of a state* q *of a Mealy machine* $\mathcal{M}$ *is the map* $[\![q]\!]^{\mathcal{M}} \colon I^* \rightharpoonup O^*$ *defined by* $[\![q]\!]^{\mathcal{M}}(\sigma) = \lambda^{\mathcal{M}}(q,\sigma)$.

States q, r *of Mealy machines* $\mathcal{M}$ *and* $\mathcal{N}$, *respectively, are* equivalent, *written* $q \approx r$, *iff* $[\![q]\!]^{\mathcal{M}} = [\![r]\!]^{\mathcal{N}}$. *Mealy machines* $\mathcal{M}$ *and* $\mathcal{N}$ *are* equivalent, *written* $\mathcal{M} \approx \mathcal{N}$, *iff their initial states are equivalent:* $q_0^{\mathcal{M}} \approx q_0^{\mathcal{N}}$. *A Mealy machine* $\mathcal{M}$ *is* minimal *iff, for all pairs of states* q, q', $q \approx q'$ *iff* $q = q'$.

▶ **Example 2.3.** Figure 1 shows an example (taken from [22]) with a graphical representation of two minimal, complete Mealy machines that are inequivalent, since the input sequence *aba* triggers different output sequences in both machines.

2.3 Observation Trees

A *functional simulation* is a function between Mealy machines that preserves the initial state and the transition and output functions.

▶ **Definition 2.4** (Simulation). *A* functional simulation *between Mealy machines* $\mathcal{M}$ *and* $\mathcal{N}$ *is a function* $f \colon Q^{\mathcal{M}} \to Q^{\mathcal{N}}$ *satisfying* $f(q_0^{\mathcal{M}}) = q_0^{\mathcal{N}}$ *and, for* $q \in Q^{\mathcal{M}}$ *and* $i \in I$,

$$\delta^{\mathcal{M}}(q,i)\!\downarrow \quad \Rightarrow \quad f(\delta^{\mathcal{M}}(q,i)) = \delta^{\mathcal{N}}(f(q),i) \text{ and } \lambda^{\mathcal{M}}(q,i) = \lambda^{\mathcal{N}}(f(q),i).$$

We write $f \colon \mathcal{M} \to \mathcal{N}$ *if* f *is a functional simulation between* $\mathcal{M}$ *and* $\mathcal{N}$.

Note that if $f \colon \mathcal{M} \to \mathcal{N}$, each transition $q \xrightarrow{i/o} q'$ of $\mathcal{M}$ can be matched by a transition $f(q) \xrightarrow{i/o} f(q')$ of $\mathcal{N}$.

For a given Mealy machine $\mathcal{M}$, an *observation tree* for $\mathcal{M}$ is a Mealy machine itself that represents the inputs and outputs that have been observed during testing of $\mathcal{M}$. Using functional simulations, we may define it formally as follows.

▶ **Definition 2.5** (Observation tree). *A Mealy machine* $\mathcal{T}$ *is a* tree *iff for each* $q \in Q^{\mathcal{T}}$ *there is a unique* $\sigma \in I^*$ *s.t.* q *is reachable via* σ. *We write* $\mathsf{access}(q)$ *for the sequence of inputs leading to* q. *For* $U \subseteq Q^{\mathcal{T}}$, *we define* $\mathsf{access}(U) = \{\mathsf{access}(q) \mid q \in U\}$. *For* $q \neq q_0^{\mathcal{T}}$, *we write* $\mathsf{parent}(q)$ *for the unique state* q' *with an outgoing transition to* q. *A tree* $\mathcal{T}$ *is an* observation tree *for a Mealy machine* $\mathcal{M}$ *iff there is a functional simulation* f *from* $\mathcal{T}$ *to* $\mathcal{M}$.

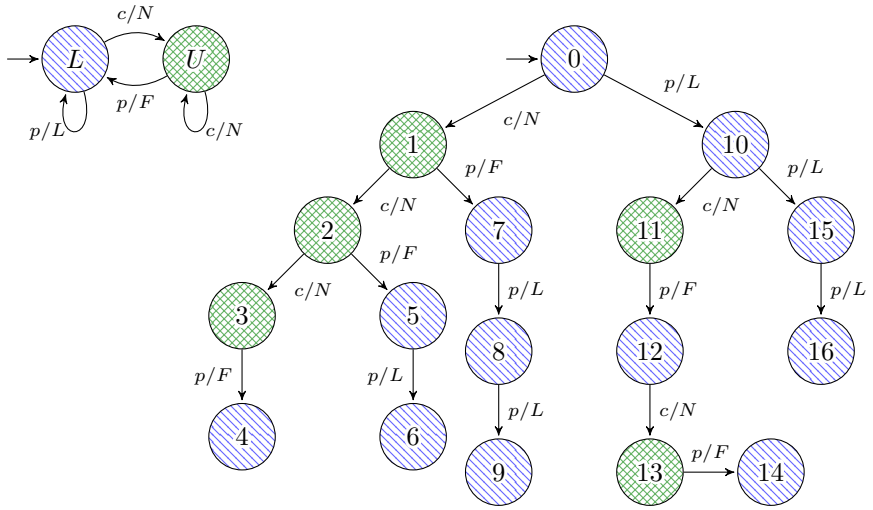


Figure 2 A Mealy machine $\mathcal{S}$ (left) and an observation tree $\mathcal{T}$ for $\mathcal{S}$ (right).

▶ **Example 2.6.** Figure 2 (right) shows an observation tree $\mathcal{T}$ for the Mealy machine $\mathcal{S}$ of Figure 2 (left). Mealy machine $\mathcal{S}$, an example taken from [32], models the behavior of a turnstile. Initially, the turnstile is locked (L), but when a coin is inserted (c) then, although no response is observed (N), the machine becomes unlocked (U). When a user pushes the bar (p) in the initial state, the turnstile is locked (L), but when the bar is pushed in the unlocked state it is free (F) and the user may pass. State colors indicate the functional simulation from $\mathcal{T}$ to $\mathcal{S}$.

2.4 Test Suites

We recall some basic vocabulary of conformance testing for Mealy machines.

▶ **Definition 2.7** (Test suites). *Let $\mathcal{S}$ be a Mealy machine. A sequence $\sigma \in I^*$ with $\delta^{\mathcal{S}}(q_0, \sigma)\downarrow$ is called a* test case *(or simply a* test*) for $\mathcal{S}$. A* test suite *T for $\mathcal{S}$ is a finite set of tests for $\mathcal{S}$. A Mealy machine $\mathcal{M}$* passes *test σ for $\mathcal{S}$ iff $\lambda^{\mathcal{M}}(q_0^{\mathcal{M}}, \sigma) = \lambda^{\mathcal{S}}(q_0^{\mathcal{S}}, \sigma)$, and passes test suite T for $\mathcal{S}$ iff it passes all tests in T.*

Observe that when $\mathcal{M}$ passes a test σ, it also passes all prefixes of σ. This means that only the maximal tests from a test suite T (tests that are not a proper prefix of another test in T) need to be executed to determine whether $\mathcal{M}$ passes T. Also note that when $\mathcal{M}$ passes a test σ, we may conclude $\delta^{\mathcal{M}}(q_0^{\mathcal{M}}, \sigma)\downarrow$.

We like to think of test suites as observation trees. Thus, for instance, the test suite $T = \{cccp, ccpp, cppp, pcpcp, ppp\}$ for the Mealy machine $\mathcal{S}$ of Figure 2(left) corresponds to the observation tree of Figure 2(right). The definition below describes the general procedure for constructing a testing tree for a given test suite T for a specification $\mathcal{S}$. The states of the testing tree are simply all the prefixes of tests in T. Since T may be empty but a tree needs to have at least one state, we require that the empty sequence ϵ is a state.

▶ **Definition 2.8** (Testing tree). *Suppose T is a test suite for a Mealy machine $\mathcal{S}$. Then the* testing tree $\mathsf{Tree}(\mathcal{S}, T)$ *is the observation tree $\mathcal{T}$ given by:*

- *$Q^{\mathcal{T}} = \{\epsilon\} \cup Pref(T)$ and $q_0^{\mathcal{T}} = \epsilon$,*
- *For all $\sigma \in I^*$ and $i \in I$ with $\sigma i \in Q^{\mathcal{T}}$, $\delta^{\mathcal{T}}(\sigma, i) = \sigma i$,*
- *For all $\sigma \in I^*$ and $i \in I$ with $\sigma i \in Q^{\mathcal{T}}$, $\lambda^{\mathcal{T}}(\sigma, i) = \lambda^{\mathcal{S}}(\delta^{\mathcal{S}}(q_0^{\mathcal{S}}, \sigma), i)$.*

There is a functional simulation from a testing tree to the specification that was used during its construction.

▶ **Lemma 2.9.** *The function f that maps each state σ of $\mathcal{T} = \mathsf{Tree}(\mathcal{S}, T)$ to the state $\delta^{\mathcal{S}}(q_0^{\mathcal{S}}, \sigma)$ of $\mathcal{S}$ is a functional simulation.*

The next lemma, which follows from the definitions, illustrates the usefulness of testing trees: a Mealy machine $\mathcal{M}$ passes a test suite T for $\mathcal{S}$ iff there exists a functional simulation from $\mathsf{Tree}(\mathcal{S}, T)$ to $\mathcal{M}$.

▶ **Lemma 2.10.** *Suppose $\mathcal{S}$ and $\mathcal{M}$ are Mealy machines, T is a test suite for $\mathcal{S}$, and $\mathcal{T} = \mathsf{Tree}(\mathcal{S}, T)$. Suppose function f maps each state σ of $\mathcal{T}$ to state $\delta^{\mathcal{M}}(q_0^{\mathcal{M}}, \sigma)$ of $\mathcal{M}$. Then $f \colon \mathcal{T} \to \mathcal{M}$ iff $\mathcal{M}$ passes T.*

3 Fault Domains and Test Suite Completeness

A *fault domain* reflects the tester's assumptions about faults that may occur in an implementation and that need to be detected during testing.

▶ **Definition 3.1** (Fault domains and $\mathcal{U}$-completeness)**.** *A* fault domain *is a set $\mathcal{U}$ of Mealy machines. A test suite T for a Mealy machine $\mathcal{S}$ is* $\mathcal{U}$-complete *if, for each $\mathcal{M} \in \mathcal{U}$, $\mathcal{M}$ passes T implies $\mathcal{M} \approx \mathcal{S}$.*

The next three lemmas are immediate consequences of the above definition. In particular, whenever T is $\mathcal{U}$-complete, completeness is preserved when we add tests to T or remove machines from $\mathcal{U}$.

▶ **Lemma 3.2.** *If T and T' are test suites with $T \subseteq T'$ and T is $\mathcal{U}$-complete, then T' is $\mathcal{U}$-complete.*

▶ **Lemma 3.3.** *If $\mathcal{U}$ and $\mathcal{U}'$ are fault domains with $\mathcal{U}' \subseteq \mathcal{U}$, and T is a $\mathcal{U}$-complete test suite, then T is $\mathcal{U}'$-complete.*

▶ **Lemma 3.4.** *If a test suite for $\mathcal{S}$ is both $\mathcal{U}$-complete and $\mathcal{U}'$-complete, then it is $\mathcal{U} \cup \mathcal{U}'$-complete.*

A particular class of fault domains that has been widely studied is based on the maximal number of states of implementations.

▶ **Definition 3.5.** *Let $m > 0$. Then fault domain $\mathcal{U}_m$ is the set of all Mealy machines with at most m states.*

In the literature, $\mathcal{U}_m$-complete test suites are usually called *m-complete*. Suppose $m \geq n$, where n is the number of states of a specification $\mathcal{S}$. Given that the size of m-complete test suites grows exponentially in $m - n$, a tester cannot possibly consider all Mealy machines with up to m states, if $m - n$ is large. Therefore, we will propose alternative and much larger fault domains that can still be fully explored during testing.

We consider fault domains of Mealy machines in which all states can be reached by first performing an access sequence from a set A (typically a state cover of specification $\mathcal{S}$), followed by k arbitrary inputs, for some small k. These fault domains capture the tester's assumption that when bugs in the implementation introduce extra states, these extra states can be reached via a few transitions from states reachable via scenarios from A. The next definition formally introduces the corresponding fault domains $\mathcal{U}_k^A$. A somewhat similar notion was previously proposed by Maarse [20], in a specific context with action refinement.

▶ **Definition 3.6.** *Let k be a natural number and let $A \subseteq I^*$. Then fault domain $\mathcal{U}_k^A$ is the set of all Mealy machines $\mathcal{M}$ such that every state of $\mathcal{M}$ can be reached by an input sequence $\sigma\rho$, for some $\sigma \in A$ and $\rho \in I^{\leq k}$.*

▶ **Example 3.7.** Mealy machine $\mathcal{M}$ from Figure 3 is contained in fault domain $\mathcal{U}_1^A$, for $A = \{\epsilon, c\}$, since all states of $\mathcal{M}$ can be reached via at most one transition from the two states L' and U' that are reachable via A.

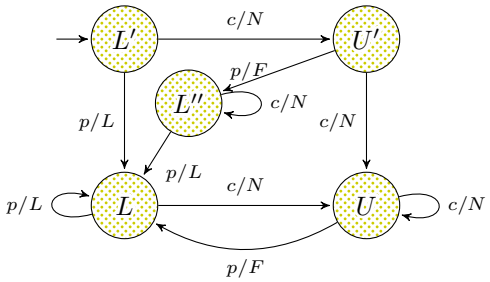


Figure 3 A Mealy machine $\mathcal{M}$ contained in fault domain $\mathcal{U}_1^A$, with $A = \{\epsilon, c\}$.

▶ Remark 3.8. The definition of $\mathcal{U}_k^A$ is closely related to the fundamental concept of *eccentricity* known from graph theory [7]. Consider a directed graph $G = (V, E)$. For vertices $w, v \in V$, let $d(w, v)$ be the length of a shortest path from w to v, or ∞ if no such a path exists. The *eccentricity* $\epsilon(w)$ of a vertex $w \in V$ is the maximum distance from w to any other vertex:

$$\epsilon(w) \quad = \quad \max_{v \in V} d(w, v).$$

This definition generalizes naturally to subsets of vertices. The distance from a set $W \subseteq V$ of vertices to a vertex $v \in V$, is the length of a shortest path from some vertex in W to v, or ∞ if no such path exists. The *eccentricity* $\epsilon(W)$ of a set of vertices W is the maximum distance from W to any other vertex:

$$d(W, v) = \min_{w \in W} d(w, v) \qquad \epsilon(W) = \max_{v \in V} d(W, v).$$

We view Mealy machines as directed graphs in the obvious way. In the example of Figure 3, the eccentricity of initial state L' is 2 (since every state of $\mathcal{M}$ can be reached with at most 2 transitions from L') and the eccentricity of state U' is ∞ (since there is no path from U' to L'). The eccentricity of $\{L', U'\}$ is 1, since state L can be reached with a single transition from L', and states L'' and U can be reached with a single transition from U'.

Fault domain $\mathcal{U}_k^A$ can alternatively be defined as the set of Mealy machines $\mathcal{M}$ for which the eccentricity of the set of states reachable via A is at most k. Note that, for a set of vertices W, $\epsilon(W)$ can be computed in linear time by contracting all elements of W to a single vertex w, followed by a breadth-first search from w.

The Mealy machine of Figure 3, which is contained in fault domain $\mathcal{U}_1^A$, has two states (L' and U') that are reached via a sequence from A, and three extra states that can be reached via a single transition from these two states. More generally, if A is a prefix closed set with n sequences and the set of inputs I contains l elements, then at most n states can be reached via sequence from A, and at most $nl - n + 1$ additional states can be reached via a single transition from states already reached by A. A second step from A may lead to $l(nl - n + 1)$ extra states, etc. This leads us to the following proposition.

▶ **Proposition 3.9.** *Let $A \subset I^*$ be prefix closed with $|A| = n$, let $|I| = l$ and $k > 0$. Then fault domain $\mathcal{U}_k^A$ contains Mealy machines with up to $n + (\sum_{j=0}^{k-1} l^j)(nl - n + 1)$ states.*

This observation implies that the number of states of Mealy machines in $\mathcal{U}_k^A$ grows exponentially in k when there are at least 2 inputs. Even for small values of k, the number of states may increase appreciably. Consider, for instance, the Mealy machine model for the Free BSD 10.2 TCP server that was obtained through black-box learning by Fiterău-Broştean et al [8]. This model has 55 states and 13 inputs, so if A is a minimal state cover, then fault domain $\mathcal{U}_2^A$ contains Mealy machines with up to 9309 states.

Even though the size of Mealy machines in $\mathcal{U}_k^A$ grows exponentially in k, fault domain $\mathcal{U}_m$ is not contained in fault domain $\mathcal{U}_k^A$ if $m = |A| + k$. For instance, the machine of Figure 2 has two states and is therefore contained in $\mathcal{U}_2$. However, this machine is not contained in $\mathcal{U}_0^A$ if we take $A = \{\epsilon, p\}$. We need to extend $\mathcal{U}_k^A$ in order to obtain a proper inclusion. Suppose that A is a minimal state cover for a minimal specification $\mathcal{S}$. Then states in $\mathcal{S}$ reached by sequences from A will be pairwise inequivalent. Methods for generating m-complete test suites typically first check whether states of implementation $\mathcal{M}$ reached by sequences from A are also inequivalent. So these methods exclude any model $\mathcal{M}$ in which distinct sequences from A reach equivalent states. This motivates the following definition:

▶ **Definition 3.10.** *Let $A \subseteq I^*$. Then fault domain $\mathcal{U}^A$ is the set of all Mealy machines $\mathcal{M}$ such that there are $\sigma, \rho \in A$ with $\sigma \neq \rho$ and $\delta^{\mathcal{M}}(q_0^{\mathcal{M}}, \sigma) \approx \delta^{\mathcal{M}}(q_0^{\mathcal{M}}, \rho)$.*

Note that $\mathcal{U}^A$ is infinite and contains Mealy machines with arbitrarily many states. Under reasonable assumptions, fault domain $\mathcal{U}_m$ is contained in fault domain $\mathcal{U}_k^A \cup \mathcal{U}^A$.

▶ **Theorem 3.11.** *Let $A \subset I^*$ be a finite set of input sequences with $\epsilon \in A$. Let k and m be natural numbers with $m = |A| + k$. Then $\mathcal{U}_m \subseteq \mathcal{U}_k^A \cup \mathcal{U}^A$.*

We refer to $\mathcal{U}_k^A \cup \mathcal{U}^A$-complete test suites as *k-A-complete*. By Theorem 3.11 and Lemma 3.3, any k-A-complete test suite is also m-complete, if $m = |A| + k$. The converse of the inclusion of Theorem 3.11 does not hold, as $\mathcal{U}^A$ may contain FSMs with an unbounded number of states. The next example illustrates that a test suite T can be m-complete, but not 0-A-complete for a state cover A of the specification.

▶ **Example 3.12.** The set $A = \{\epsilon, a, aa\}$ is a minimal state cover of Mealy machine $\mathcal{S}$ at the left of Figure 4. Machine $\mathcal{S}$ is minimal and *aaa* is a *distinguishing sequence* since it generates different outputs for each of the three states of $\mathcal{S}$. Mealy machine $\mathcal{S}$ is trivially contained in fault domains $\mathcal{U}_3$ and $\mathcal{U}_0^A$, whereas Mealy machine $\mathcal{M}$ at the right of Figure 4 is clearly not contained in $\mathcal{U}_3$ and $\mathcal{U}_0^A$.

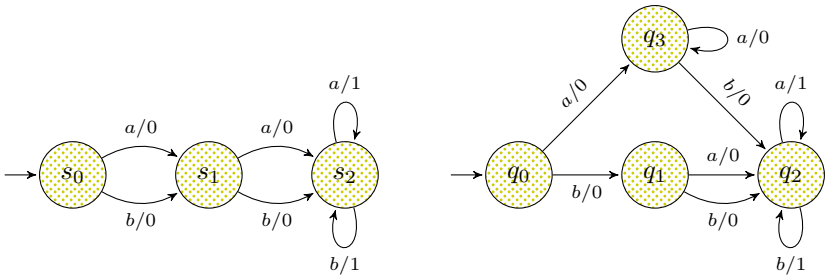


Figure 4 A specification $\mathcal{S}$ (left) and an inequivalent implementation $\mathcal{M}$ (right).

However, $\mathcal{M}$ is contained in fault domain $\mathcal{U}^A$ since $\delta^{\mathcal{M}}(q_0, a) = \delta^{\mathcal{M}}(q_0, aa)$. Note that the set $B = \{\epsilon, b, bb\}$ is also a minimal state cover for $\mathcal{S}$, and the sequence bbb is also a distinguishing sequence. Using the Wp-method, to be discussed in more detail in Section 5, 0-B-complete test suites can easily be built from a state cover and a distinguishing sequence. In particular, the set $T = B \cdot \{bbb\} \quad \cup \quad B \cdot \{a, b\} \cdot \{bbb\}$ is a 0-B-complete test suite for $\mathcal{S}$, and therefore (by Theorem 3.11) also 3-complete. However, since $\mathcal{M}$ passes test suite T, T is not $\mathcal{U}_0^A \cup \mathcal{U}^A$-complete, and thus not 0-A-complete.

Below we give an example to show that, for $k > 0$, m-complete test suites generated by the SPYH-method [32] are not always k-A-complete, if $m = |A| + k$. Appendix B contains variations of this example, which demonstrate that the SPY-method [30] and the H-method [6] are not k-A-complete either.

▶ **Example 3.13.** Consider specification $\mathcal{S}$ and testing tree $\mathcal{T}$ from Figure 2. This specification and the corresponding test suite $T = \{cccp, ccpp, cppp, pcpcp, ppp\}$ were both taken from [32], where the SPYH-method was used to generate T, which was shown to be 3-complete for $\mathcal{S}$. Consider the minimal state cover $A = \{\epsilon, c\}$ for $\mathcal{S}$. FSM $\mathcal{M}$ from Figure 3 belongs to fault domain $\mathcal{U}_1^A$, since all states can be reached via at most one transition from L' or U'. Clearly $\mathcal{S} \not\approx \mathcal{M}$, as input sequence $cpcp$ provides a counterexample. Nevertheless, $\mathcal{M}$ passes test suite T. Thus the test suite generated by the SPYH-method [32] is not 1-A-complete.

4 A Sufficient Condition for k-A-Completeness

In this section, we describe a sufficient condition for a test suite to be k-A-complete, which (based on ideas of [6, 35, 37]) is phrased entirely in terms of properties of its testing tree. This tree should contain access sequences for each state in the specification, successors for these states for all possible inputs should be present up to depth $k + 1$, and apartness relations between certain states of the tree should hold. Before we present our condition and its correctness proof, we first need to introduce the concepts of apartness, basis and stratification, and study their basic properties.

4.1 Apartness

In our sufficient condition, the concept of *apartness*, inspired by a similar notion that is standard in constructive real analysis [33, 11], plays a central role.

▶ **Definition 4.1** (Apartness). *For a Mealy machine $\mathcal{M}$, we say that states $q, r \in Q^{\mathcal{M}}$ are* apart *(written $q \# r$) iff there is some $\sigma \in I^*$ such that $[\![q]\!](\sigma)\downarrow$, $[\![r]\!](\sigma)\downarrow$, and $[\![q]\!](\sigma) \neq [\![r]\!](\sigma)$. We say that σ is a* separating sequence *for q and r. We also call σ a* witness *of $q \# r$ and write $\sigma \vdash q \# p$.*

Note that the apartness relation $\# \subseteq Q \times Q$ is irreflexive and symmetric. For the observation tree of Figure 2 we may derive the following apartness pairs and corresponding witnesses: $p \vdash 0 \# 1$ and $p \vdash 0 \# 11$. Observe that when two states are apart they are not equivalent, but states that are not equivalent are not necessarily apart. States 0 and 12, for instance, are neither equivalent nor apart. However, for complete Mealy machines apartness coincides with inequivalence.

The apartness of states $q \# r$ expresses that there is a conflict in their semantics, and consequently, apart states can never be identified by a functional simulation.

▶ **Lemma 4.2.** *For a functional simulation $f : \mathcal{T} \to \mathcal{M}$,*

$$q \# r \text{ in } \mathcal{T} \quad \Longrightarrow \quad f(q) \# f(r) \text{ in } \mathcal{M} \qquad \text{for all } q, r \in Q^{\mathcal{T}}.$$

Thus, whenever states are apart in the observation tree $\mathcal{T}$, we know that the corresponding states in Mealy machine $\mathcal{M}$ are distinct. The apartness relation satisfies a weaker version of *co-transitivity*, stating that if $\sigma \vdash r \# r'$ and q has the transitions for σ, then q must be apart from at least one of r and r', or maybe even both.

▶ **Lemma 4.3** (Weak co-transitivity)**.** *In every Mealy machine $\mathcal{M}$,*

$$\sigma \vdash r \# r' \ \wedge \ \delta(q, \sigma)\downarrow \ \Longrightarrow \ r \# q \ \vee \ r' \# q \qquad \text{for all } r, r', q \in Q^{\mathcal{M}}, \sigma \in I^*.$$

4.2 Basis

In each observation tree, we may identify a *basis*: an ancestor closed set of states that are pairwise apart. In general, a basis is not uniquely determined, and an observation tree may for instance have different bases with the same size. However, once we have fixed a basis, the remaining states in the tree can be uniquely partitioned by looking at their distance from this basis.

▶ **Definition 4.4** (Basis)**.** *Let $\mathcal{T}$ be an observation tree. A nonempty subset of states $B \subseteq Q^{\mathcal{T}}$ is called a* basis *of $\mathcal{T}$ if*
1. *B is ancestor-closed: for all $q \in B : q \neq q_0^{\mathcal{T}} \ \Rightarrow \ \mathsf{parent}(q) \in B$, and*
2. *states in B are pairwise apart: for all $q, q' \in B : q \neq q' \ \Rightarrow \ q \# q'$.*
For each state q of $\mathcal{T}$, the candidate set *$C(q)$ is the set of basis states that are not apart from q: $C(q) \ = \ \{q' \in B \mid \neg(q \# q')\}$. State q is* identified *if $|C(q)| = 1$.*

Since B is nonempty and ancestor-closed, all states on the access path of a basis state are in the basis as well. In particular, the initial state $q_0^{\mathcal{T}}$ is in the basis. Also note that, by definition, basis states are identified. The next lemma gives some useful properties of a basis.

▶ **Lemma 4.5.** *Suppose $\mathcal{T}$ is an observation tree for $\mathcal{M}$ with $f : \mathcal{T} \to \mathcal{M}$ and basis B such that $|B| = |Q^{\mathcal{M}}|$. Then f restricted to B is a bijection, $\mathcal{M}$ is minimal, and $\mathsf{access}(B)$ is a minimal state cover for $\mathcal{M}$.*

Whenever a subset of states B of a testing tree is a basis, the corresponding test suite is $\mathcal{U}^A$-complete, for $A = \mathsf{access}(B)$.

▶ **Lemma 4.6.** *Let $\mathcal{S}$ be a Mealy machine, let T be a test suite for $\mathcal{S}$, let B be a basis for $\mathcal{T} = \mathsf{Tree}(\mathcal{S}, T)$, and let $A = \mathsf{access}(B)$. Then T is $\mathcal{U}^A$-complete.*

4.3 Stratification

A basis B induces a stratification of observation tree $\mathcal{T}$: first we have the set F^0 of immediate successors of basis states that are not basis states themselves, next the set F^1 of immediate successors of states in F^0, etc. In general, F^k contains all states that can be reached via a path of length $k + 1$ from B.

▶ **Definition 4.7** (Stratification)**.** *Let $\mathcal{T}$ be an observation tree with basis B. Then B induces a* stratification *of $Q^{\mathcal{T}}$ as follows. For $k \geq 0$,*

$$F^k \ = \ \{q \in Q^{\mathcal{T}} \mid d(B, q) = k + 1\}.$$

*We call F^k the k-*level frontier *and write $F^{<k} = \bigcup_{0 \leq i < k} F^i$ and $F^{\leq k} = \bigcup_{0 \leq i \leq k} F^i$.*

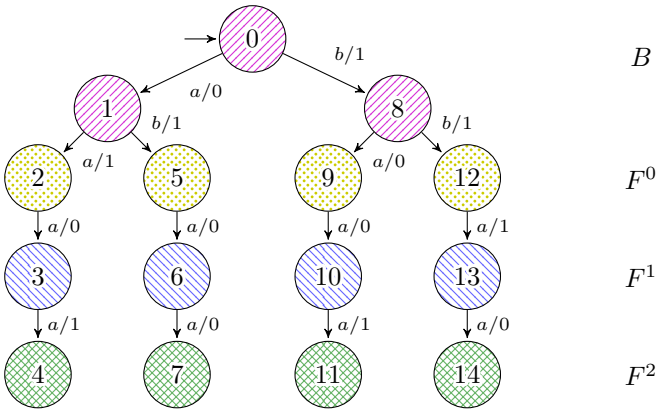


Figure 5 Stratification of an observation tree induced by $B = \{0, 1, 8\}$.

▶ **Example 4.8.** Figure 5 shows the stratification for an observation tree for specification $\mathcal{S}$ from Figure 1 induced by basis $B = \{0, 1, 8\}$. Witness aa shows that the three basis states are pairwise apart, and therefore identified. States from sets B, F^0, F^1 and F^2 are marked with different colors. In Figure 5, B is complete, but F^0, F^1 and F^2 are incomplete (since states of F^0 and F^1 have no outgoing b-transitions, and states of F^2 have no outgoing transitions at all). The four F^0 states are also identified since $C(2) = \{0\}$, $C(5) = \{8\}$, $C(9) = \{0\}$, and $C(12) = \{1\}$. Two states in F^1 are identified since $C(3) = C(10) = \{1\}$, whereas the other two are not since $C(6) = C(13) = \{0, 8\}$. Since states in F^2 have no outgoing transitions, they are not apart from any other state, and thus $C(4) = C(7) = C(11) = C(14) = \{0, 1, 8\}$.

4.4 A Sufficient Condition for k-A-completeness

We are now prepared to state our characterization theorem.

▶ **Theorem 4.9.** *Let $\mathcal{M}$ and $\mathcal{S}$ be Mealy machines, let $\mathcal{T}$ be an observation tree for both $\mathcal{M}$ and $\mathcal{S}$, let B be a basis for $\mathcal{T}$ with $|B| = |Q^{\mathcal{S}}|$, let $A = \mathsf{access}(B)$, let $F^0, F^1, \ldots$ be the stratification induced by B, and let $k \geq 0$. Suppose B and $F^{<k}$ are complete, all states in F^k are identified, and the following condition holds:*

$$\forall q \in F^k \; \forall r \in F^{<k} : \qquad C(q) = C(r) \vee q \# r \tag{1}$$

Suppose that $\mathcal{M} \in \mathcal{U}_k^A$. Then $\mathcal{S} \approx \mathcal{M}$.

As a corollary, we obtain a sufficient condition for k-A-completeness.

▶ **Corollary 4.10.** *Let $\mathcal{S}$ be a Mealy machine, let T be a test suite for $\mathcal{S}$, let $\mathcal{T} = \mathsf{Tree}(\mathcal{S}, T)$, let B be a basis for $\mathcal{T}$ with $|B| = |Q^{\mathcal{S}}|$, let $A = \mathsf{access}(B)$, let $F^0, F^1, \ldots$ be the stratification of $\mathcal{T}$ induced by B, and let $k \geq 0$. Suppose B and $F^{<k}$ are complete, all states in F^k are identified, and condition (1) holds. Then T is k-A-complete.*

The conditions of Corollary 4.10 do not only impose restrictions on testing tree $\mathcal{T}$, but also on specification $\mathcal{S}$. Suppose that the conditions of Corollary 4.10 hold. Then, by Lemma 2.9, there is a functional simulation $f : \mathcal{T} \to \mathcal{S}$. By Lemma 4.5, f restricted to B is a bijection, $\mathcal{S}$ is minimal, and $\mathsf{access}(B)$ is a minimal state cover for $\mathcal{S}$. Furthermore, since B is complete and f restricted to B is a bijection, $\mathcal{S}$ is also complete. Minimality and completeness of specifications are common assumptions in conformance testing.

▶ **Example 4.11.** A simple example of the application of Corollary 4.10, is provided by the observation tree from Figure 5 for the specification $\mathcal{S}$ from Figure 1. This observation tree corresponds to the test suite $T = \{aaaa, abaa, baaa, bbaa\}$. We claim that this test suite is 0-A-complete, for $A = \{\epsilon, a, b\} = \mathsf{access}(B)$. Note that condition (1) vacuously holds when $k = 0$. All other conditions of Corollary 4.10 are also met: basis B is complete and all states in F^0 are identified. Therefore, test suite T is 0-A-complete. We may slightly optimize the test suite by replacing test *bbaa* by test *bba*, since all conditions of the corollary are still met for the reduced testing tree.

The conditions of Corollary 4.10 are sufficient for k-A-completeness, but the following trivial example illustrates that they are not necessary.

▶ **Example 4.12.** Consider the Mealy machines $\mathcal{S}$ of Figure 6, which has a single state, two inputs a and b, and minimal state cover $A = \{\epsilon\}$. Test suite $T = \{ab\}$ does not meet the

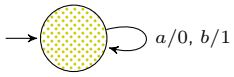


Figure 6 Test suite $T = \{ab\}$ is 0-$\{\epsilon\}$-complete for the above specification.

conditions of Corollary 4.10, since the basis of the corresponding testing tree is not complete. Nevertheless, we claim that T is 0-A-complete. Since A is a singleton, the fault domain $\mathcal{U}^A$ is empty. The fault domain $\mathcal{U}_0^A$ only contains Mealy machines with a single state, and self-loop transitions for inputs a and b. Test suite T verifies that the outputs of these transitions are in agreement with $\mathcal{S}$. Thus, any machine in $\mathcal{U}_0^A$ that passes T will be equivalent to $\mathcal{S}$. Hence T is 0-A-complete.

Theorem 4.9 and Corollary 4.10 do not require that all states in $F^{<k}$ are identified, but the other conditions of the theorem/corollary already imply this.

▶ **Proposition 4.13.** *Let $\mathcal{T}$ be an observation tree for $\mathcal{S}$, B a basis for $\mathcal{T}$ with $|B| = |Q^{\mathcal{S}}|$, $F^0, F^1, \ldots$ the stratification induced by B, and $k \geq 0$. Suppose B and $F^{<k}$ are complete, all states in F^k are identified, and condition (1) holds. Then all states in $F^{<k}$ are identified*

Appendix A contains an example to illustrate that the converse implication does not hold: even if all states in $B \cup F^{\leq k}$ are identified, condition (1) may not hold. The next proposition asserts that condition (1) implies co-transitivity for a much larger collection of triples, namely triples of a basis state, a state in F^i and a state in F^j, for all $i \neq j$.

▶ **Proposition 4.14.** *Let $\mathcal{T}$ be an observation tree for $\mathcal{S}$, B a basis for $\mathcal{T}$ with $|B| = |Q^{\mathcal{S}}|$, $F^0, F^1, \ldots$ the stratification induced by B, and $k \geq 0$. Suppose B and $F^{<k}$ are complete, all states in F^k are identified, and condition (1) holds. Then $\forall i, j\ \forall q \in F^i\ \forall r \in F^j : 0 \leq i < j \leq k \quad \Rightarrow \quad C(q) = C(r) \vee q \# r$*

As a consequence of the next proposition, condition (1) can be equivalently formulated as

$$\forall q \in F^k\ \forall r \in F^{<k}\ \forall s \in B : s \# q \quad \Rightarrow \quad s \# r \vee q \# r \tag{2}$$

Condition (2) says that apartness is *co-transitive* for triples of states in the observation tree consisting of a state in F^k, a state in $F^{<k}$, and a basis state. Co-transitivity is a fundamental property of apartness [33, 11].

▶ **Proposition 4.15.** *Let $\mathcal{T}$ be an observation tree for $\mathcal{S}$, B a basis for $\mathcal{T}$, and $|B| = |Q^{\mathcal{S}}|$. Suppose q and r are states of $\mathcal{T}$ and q is identified. Then*

$$C(q) = C(r) \vee q \# r \quad \Leftrightarrow \quad [\forall s \in B : s \# q \ \Rightarrow\ s \# r \vee q \# r]$$

4.5 Algorithm

We will now present an algorithm that checks, for a given observation tree $\mathcal{T}$ with N states, in $\Theta(N^2)$ time, for all pairs of states, whether they are apart or not. In practice, models of realistic protocols often have up to a few dozen states and inputs [24]. This means that if k equals 2 or 3, the observation tree will contain up to a few thousand states, and so with some optimization (e.g. on the fly computation of apartness pairs) our algorithm is practical. For larger benchmarks the observation trees already contain millions of states, and our algorithm cannot be applied. Nevertheless, from a theoretical perspective it is interesting to know that the apartness relation (and hence also our sufficient condition for k-A-completeness) can be computed in polynomial time.

Algorithm 1 Computing the apartness relation.

```
1: function FillApartnessArray( )
2:     for q ∈ Q do
3:         for q′ ∈ Q do
4:             if ¬Visited(q, q′) then ApartnessCheck(q, q′)
5:             end if
6:         end for
7:     end for
8:     return Apart
9: end function
10: function ApartnessCheck(q, q′)
11:     l ← Adj[q], l′ ← Adj[q′]
12:     while l ≠ ϵ ∧ l′ ≠ ϵ ∧ ¬Apart(q, q′) do
13:         r ← hd(l), r′ ← hd(l′)
14:         if in(r) < in(r′) then l ← tl(l)
15:         else if in(r′) < in(r) then l′ ← tl(l′)
16:         else if out(r) = out(r′) then
17:             if ¬Visited(r, r′) then ApartnessCheck(r, r′)
18:             end if
19:             Apart(q, q′) ← Apart(q, q′) ∨ Apart(r, r′)
20:             l ← tl(l), l′ ← tl(l′)
21:         elseApart(q, q′) ← true
22:         end if
23:     end while
24:     Visited(q, q′) ← true
25: end function
```

Algorithm 1 assumes a total order $<$ on the set of inputs I and two partial functions $in : Q^{\mathcal{T}} \rightharpoonup I$ and $out : Q^{\mathcal{T}} \rightharpoonup O$ which, for each noninitial state q, specify the input and output, respectively, of the unique incoming transition of q. It also assumes, for each state $q \in Q^{\mathcal{T}}$, an adjacency list $Adj[q] \in (Q^{\mathcal{T}})^*$ that contains the immediate successors of q, sorted on their inputs. The algorithm maintains two Boolean arrays *Visited* and *Apart* to record whether a pair of states (q, q') has been visited or is apart, respectively. Initially, all entries in both arrays are *false*. When exploring a pair of states (q, q'), Algorithm 1 searches for outgoing transitions of q and q' with the same input label. In this case, if the outputs are different then it concludes that q and q' are apart. Otherwise, if the outputs are the same, it considers the target states r and r'. If the pair (r, r') has not been visited yet then the algorithm recursively explores whether this pair of states is apart. If r and r' are apart then q and q' are apart as well.

The theorem below asserts the correctness and time complexity of Algorithm 1.

▶ **Theorem 4.16.** *Algorithm 1 terminates with running time* $\Theta(N^2)$*, where* $N = |Q^{\mathcal{T}}|$*. Upon termination,* $Apart(q, q') = true$ *iff* $q \# q'$*, for all* $q, q' \in Q^{\mathcal{T}}$*.*

Once we know the apartness relation, there are several things we can do, e.g., we may check in $\Theta(N^2)$ time whether the conditions of Theorem 4.9 hold (but note that N grows exponentially in k). First we check in linear time whether all states in B and $F^{<k}$ are complete. Next, in one pass over array *Apart*, we compute the candidate sets for all frontier states. By Proposition 4.13, if some state in the frontier has a candidate set with more than one element, then the conditions of Theorem 4.9 do not hold. Otherwise, we can check in constant time whether frontier states have the same candidate set. Finally, we check condition (1) with a double for-loop over F^k and $F^{<k}$.

If we can compute the apartness relation, we may also select appropriate state identifiers on-the-fly in order to generate small k-A-complete test suites (similar to the H-method [6]), and we can check in $\Theta(N^2)$ time whether tests can be removed from a given test suite without compromising k-A-completeness.

5 Deriving Completeness for Existing Methods

In this section, we show how k-A-completeness of two popular algorithms for test suite generation, the Wp and HSI methods, follows from Corollary 4.10. We also present an alternative m-completeness proof of the H-method [6], which is a minor variation of the proof of Theorem 4.9. In order to define the Wp and HSI methods, we need certain sets (of sets) of sequences. The following definitions are based on [22]; see [5, 22] for a detailed exposition.

▶ **Definition 5.1.** *Let* $\mathcal{S}$ *be a Mealy machine.*

- *A* state identifier *for a state* $q \in Q^{\mathcal{S}}$ *is a set* $W_q \subseteq I^*$ *such that for every inequivalent state* $r \in Q^{\mathcal{S}}$*,* W_q *contains a separating sequence for* q *and* r*, i.e., a witness for their apartness.*
- *We write* $\{W_q\}_{q \in Q^{\mathcal{S}}}$ *or simply* $\{W_q\}_q$ *for a set that contains a state identifier* W_q *for each* $q \in Q^{\mathcal{S}}$*.*
- *If* $\mathcal{W} = \{W_q\}_q$ *is a set of state identifiers, then the* flattening $\bigcup \mathcal{W}$ *is the set* $\{\sigma \in I^* \mid \exists q \in Q^{\mathcal{S}} : \sigma \in W_q\}$*.*
- *If* W *is a set of input sequences and* $\mathcal{W} = \{W_q\}_q$ *is a set of state identifiers, then the* concatenation $W \odot \mathcal{W}$ *is defined as* $\{\sigma\tau \mid \sigma \in W,\ \tau \in W_{\delta^{\mathcal{S}}(q_0^{\mathcal{S}}, \sigma)}\}$*.*
- *A set of state identifiers* $\{W_q\}_q$ *is* harmonized *if, for each pair of inequivalent states* $q, r \in Q^{\mathcal{S}}$*,* $W_q \cap W_r$ *contains a separating sequence for* q *and* r*. We refer to such a set of state identifiers as a* separating family*.*

k-A-completeness of the Wp-method [9] follows from Corollary 4.10 via routine checking.

▶ **Proposition 5.2** (k-A-completeness of the Wp-method)**.** *Let* $\mathcal{S}$ *be a complete, minimal Mealy machine,* $k \geq 0$*,* A *a minimal state cover for* $\mathcal{S}$*, and* $\mathcal{W} = \{W_q\}_q$ *a set of state identifiers. Then* $T = (A \cdot I^{\leq k+1}) \cup (A \cdot I^{\leq k} \cdot \bigcup \mathcal{W}) \cup (A \cdot I^{\leq k+1} \odot \mathcal{W})$ *is a* k*-*A*-complete test suite for* $\mathcal{S}$*.*

Also k-A-completeness of the HSI-method of Luo et al [19] and Petrenko et al [41, 28] follows from Corollary 4.10 via a similar argument.

▶ **Proposition 5.3** (k-A-completeness of the HSI-method)**.** *Let* $\mathcal{S}$ *be a complete, minimal Mealy machine,* $k \geq 0$*,* A *a minimal state cover for* $\mathcal{S}$*, and* $\mathcal{W}$ *a separating family. Then* $T = (A \cdot I^{\leq k+1}) \cup (A \cdot I^{\leq k+1} \odot \mathcal{W})$ *is* k*-*A*-complete for* $\mathcal{S}$*.*

The W-method of [39, 3], and the UIOv-method of [2] are instances of the Wp-method, and the ADS-method of [17] and the hybrid ADS method of [31] are instances of the HSI-method. This means that k-A-completeness of these methods also follows from our results.

The H-method of Dorofeeva et al [6] is based on a variant of our Theorem 4.9 which requires that all states in $F^{\leq k}$ are identified and replaces condition (1) by

$$\forall q, r \in F^{\leq k} : q \xrightarrow{+} r \quad \Rightarrow \quad C(q) = C(r) \vee q \# r \tag{3}$$

By Proposition 4.14, condition (3) is implied by condition (1) of Theorem 4.9. Appendix B contains an example showing that the H-method is not k-A-complete. However, as shown by [6, Theorem 1], the H-method is m-complete. Our condition (1) can be viewed as a strengthening of condition (3) of the H-method, needed for k-A-completeness. Below we present an alternative formulation of the m-completeness result for the H-method of [6], restated for our setting. The full version of this article contains a proof of this proposition that uses the same proof technique as Theorem 4.9.

▶ **Proposition 5.4** (m-completeness of the H-method). *Let $\mathcal{S}$ be a Mealy machine with n states, let T be a test suite for $\mathcal{S}$, let $\mathcal{T} = \mathsf{Tree}(\mathcal{S}, T)$, let B be a basis for $\mathcal{T}$ with n states, let $A = \mathsf{access}(B)$, let $F^0, F^1, \ldots$ be the stratification of $\mathcal{T}$ induced by B, and let $m, k \geq 0$ with $m = n + k$. Suppose B and $F^{<k}$ are complete, all states in $F^{\leq k}$ are identified, and condition (3) holds. Then T is m-complete.*

6 Conclusions and Future Work

In order to solve a long-standing open problem of Hennie [12], we proposed the notion of k-A-completeness. We showed that the fault domain for k-A-completeness is larger than the one for m-completeness (if $m = |A| + k$), and includes FSMs with a number of extra states that grows exponentially in k. We provided a sufficient condition for k-A-completeness in terms of apartness of states of the observation induced by a test suite. Our condition can be checked efficiently (in terms of the size of the test suite) and can be used to prove k-A-completeness of the Wp-method of [9] and the HSI-method of [19, 41, 28]. Our results show that the Wp and HSI methods are complete for much larger fault domains than the ones for which they were originally designed. We presented counterexamples to show that the SPY-method [30], the H-method [6], and the SPYH-method [32] are not k-A-complete.

We view our results as an important step towards the definition of fault domains that capture realistic assumptions about possible faults, but still allow for the design of sufficiently small test suites that are complete for the fault domain and can be run within reasonable time. A promising research direction is to reduce the size of the fault domains via reasonable assumptions on the structure of the implementation under test. Kruger et al [16] show that significantly smaller test suites suffice if the fault domains $\mathcal{U}_m$ are reduced by making plausible assumptions about the implementation. The same approach can also be applied for the fault domains $\mathcal{U}_k^A \cup \mathcal{U}^A$ proposed in this article. We expect that our results will find applications in the area of model learning [1, 25, 34, 13]. Black-box conformance testing, which is used to find counterexamples for hypothesis models, has become the main bottleneck in applications of model learning [34, 40]. This provides motivation and urgency to find fault domains that provide a better fit with applications and allow for smaller test suites [16].

An important question for future research is to explore empirical evidence for the usefulness of our new fault domains: how often are faulty implementations of real systems contained in fault domains $\mathcal{U}_k^A$, for small k and a sensible choice of A? Hübner et al [14] investigated how test generation methods perform for SUTs whose behaviors lie outside the fault domain.

They considered some realistic SUTs – SystemC implementations of a ceiling speed monitor and an airbag controller described in SysML – and used mutation operators to introduce bugs in a systematic and automated manner. Their experiments show that m-complete test suites generated by the W- and Wp-methods exhibit significantly greater test strength than conventional random testing, even for behavior outside the fault domain. It would be interesting to revisit these experiments and check which fraction of the detected faults is outside $\mathcal{U}_m$ but contained in $\mathcal{U}_k^A$.

Our condition is sufficient but not necessary for completeness. If one can prove, based on the assumptions of the selected fault domain, that two traces σ and τ reach the same state both in specification $\mathcal{S}$ and in implementation $\mathcal{M}$, then it makes no difference in test suites whether a suffix ρ is appended after σ or after τ. Simão et al [4, 30] were the first to exploit this idea of *convergence* to reduce the size of test suites in a setting for m-completeness. It is future work to adapt these results to reduce the size of k-A-complete test suites. Also, the complexity of deciding whether a test suite is k-A-complete is still unknown. Another direction for future work is to lift our results to partial FSMs with observable nondeterminism, e.g. by adapting the state counting method [28].

Closest to our characterization is the work of Sachtleben [29], who develops unifying frameworks for proving m-completeness of test generation methods. Inspired by the H-method, he defines an abstract H-condition that is sufficiently strong to prove m-completeness of the Wp, HSI, SPY, H and SPYH methods. Sachtleben [29] also considers partially defined FSMs with observable nondeterminism, and takes convergence into account. Moreover, he mechanized all his proofs using Isabelle. It would be interesting to explore whether the formalization of [29] can be adapted to our notion of k-A-completeness.

An obvious direction for future work is to extend our notion of k-A-completeness to richer classes of models, such as timed automata and register automata. The recent work of [15] may provide some guidance here. It will also be interesting to explore if our characterization can be used for efficient test suite generation, or for pruning test suites that have been generated by other methods.

References

1 D. Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987. doi:10.1016/0890-5401(87)90052-6.

2 W. Y. L. Chan, C. T. Vuong, and M. R. Otp. An improved protocol test generation procedure based on UIOs. In *Proceedings of the ACM Symposium on Communications Architectures & Protocols, September 19-22, 1989*, SIGCOMM '89, pages 283–294, New York, NY, USA, 1989. Association for Computing Machinery. doi:10.1145/75246.75274.

3 T.S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, 4(3):178–187, 1978. doi:10.1109/TSE.1978.231496.

4 Adenilso da Silva Simão, Alexandre Petrenko, and Nina Yevtushenko. Generating reduced tests for FSMs with extra states. In Manuel Núñez, Paul Baker, and Mercedes G. Merayo, editors, *Testing of Software and Communication Systems, 21st IFIP WG 6.1 International Conference, TESTCOM 2009 and 9th International Workshop, FATES 2009, Eindhoven, The Netherlands, November 2-4, 2009. Proceedings*, volume 5826 of *Lecture Notes in Computer Science*, pages 129–145. Springer, 2009. doi:10.1007/978-3-642-05031-2_9.

5 Rita Dorofeeva, Khaled El-Fakih, Stéphane Maag, Ana R. Cavalli, and Nina Yevtushenko. FSM-based conformance testing methods: A survey annotated with experimental evaluation. *Information & Software Technology*, 52(12):1286–1297, 2010. doi:10.1016/j.infsof.2010.07.001.

6 Rita Dorofeeva, Khaled El-Fakih, and Nina Yevtushenko. An improved conformance testing method. In Farn Wang, editor, *Formal Techniques for Networked and Distributed Systems - FORTE 2005, 25th IFIP WG 6.1 International Conference, Taipei, Taiwan, October 2-5, 2005, Proceedings*, volume 3731 of *Lecture Notes in Computer Science*, pages 204–218. Springer, 2005. doi:10.1007/11562436_16.

7 Feodor F. Dragan and Arne Leitert. On the minimum eccentricity shortest path problem. *Theoretical Computer Science*, 694:66–78, 2017. doi:10.1016/j.tcs.2017.07.004.

8 P. Fiterău-Broştean, R. Janssen, and F.W. Vaandrager. Combining model learning and model checking to analyze TCP implementations. In S. Chaudhuri and A. Farzan, editors, *Proceedings 28th International Conference on Computer Aided Verification (CAV'16),* Toronto, Ontario, Canada, volume 9780 of *Lecture Notes in Computer Science*, pages 454–471. Springer, 2016. doi:10.1007/978-3-319-41540-6_25.

9 S. Fujiwara, G. von Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite state models. *IEEE Transactions on Software Engineering*, 17(6):591–603, 1991. doi:10.1109/32.87284.

10 Maciej Gazda and Robert M. Hierons. Model independent refusal trace testing. *Science of Computer Programming*, 239:103173, 2025. doi:10.1016/j.scico.2024.103173.

11 Herman Geuvers and Bart Jacobs. Relating apartness and bisimulation. *Logical Methods in Computer Science*, Volume 17, Issue 3, July 2021. doi:10.46298/lmcs-17(3:15)2021.

12 F. C. Hennie. Fault detecting experiments for sequential circuits. In *Proceedings of the Fifth Annual Symposium on Switching Circuit Theory and Logical Design*, pages 95–110, 1964. doi:10.1109/SWCT.1964.8.

13 Falk Howar and Bernhard Steffen. Active automata learning in practice. In Amel Bennaceur, Reiner Hähnle, and Karl Meinke, editors, *Machine Learning for Dynamic Software Analysis: Potentials and Limits: International Dagstuhl Seminar 16172, Dagstuhl Castle, Germany, April 24-27, 2016, Revised Papers*, pages 123–148. Springer International Publishing, 2018.

14 Felix Hübner, Wen-ling Huang, and Jan Peleska. Experimental evaluation of a novel equivalence class partition testing strategy. *Softw. Syst. Model.*, 18(1):423–443, 2019. doi:10.1007/S10270-017-0595-8.

15 Bálint Kocsis and Jurriaan Rot. Complete test suites for automata in monoidal closed categories. In Parosh Aziz Abdulla and Delia Kesner, editors, *Foundations of Software Science and Computation Structures - 28th International Conference, FoSSaCS 2025, Held as Part of the International Joint Conferences on Theory and Practice of Software, ETAPS 2025, Hamilton, ON, Canada, May 3-8, 2025, Proceedings*, volume 15691 of *Lecture Notes in Computer Science*, pages 198–219. Springer, 2025. doi:10.1007/978-3-031-90897-2_10.

16 Loes Kruger, Sebastian Junges, and Jurriaan Rot. Small test suites for active automata learning. In Bernd Finkbeiner and Laura Kovács, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 30th International Conference, TACAS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part II*, volume 14571 of *Lecture Notes in Computer Science*, pages 109–129. Springer, 2024. doi:10.1007/978-3-031-57249-4_6.

17 D. Lee and M. Yannakakis. Testing finite-state machines: State identification and verification. *IEEE Transactions on Computers*, 43(3):306–320, 1994. doi:10.1109/12.272431.

18 D. Lee and M. Yannakakis. Principles and methods of testing finite state machines — a survey. *Proceedings of the IEEE*, 84(8):1090–1123, 1996. doi:10.1109/5.533956.

19 Gang Luo, Alexandre Petrenko, and Gregor von Bochmann. Selecting test sequences for partially-specified nondeterministic finite state machines. In Tadanori Mizuno, Teruo Higashino, and Norio Shiratori, editors, *Protocol Test Systems: 7th workshop 7th IFIP WG 6.1 international workshop on protocol text systems*, pages 95–110, Boston, MA, 1995. Springer US. doi:10.1007/978-0-387-34883-4_6.

20 T. Maarse. *Active Mealy Machine Learning using Action Refinements.* Master thesis, Radboud University Nijmegen, August 2020. URL: https://www.cs.ru.nl/F.Vaandrager/thesis-2019_NWI-IMC048_s4416295.pdf.

21 I. Melse. *Sufficient Conditions for k-Completeness Using Observation Trees and Apartness.* Bachelor thesis, ICIS, Radboud University Nijmegen, April 2024. URL: https://www.cs.ru.nl/bachelors-theses/2024/Ivo_Melse___s1088677___Sufficient_Conditions_for_k-completeness_using_Observation_Trees_and_Apartness.pdf.

22 J. Moerman. *Nominal Techniques and Black Box Testing for Automata Learning.* PhD thesis, Radboud University Nijmegen, July 2019.

23 E.F. Moore. Gedanken-experiments on sequential machines. In *Automata Studies*, volume 34 of *Annals of Mathematics Studies*, pages 129–153. Princeton University Press, 1956.

24 Daniel Neider, Rick Smetsers, Frits W. Vaandrager, and Harco Kuppens. Benchmarks for automata learning and conformance testing. In Tiziana Margaria, Susanne Graf, and Kim G. Larsen, editors, *Models, Mindsets, Meta: The What, the How, and the Why Not?*, volume 11200 of *Lecture Notes in Computer Science*, pages 390–416. Springer, 2018. doi:10.1007/978-3-030-22348-9_23.

25 Doron Peled, Moshe Y. Vardi, and Mihalis Yannakakis. Black box checking. In Jianping Wu, Samuel T. Chanson, and Qiang Gao, editors, *Proceedings FORTE*, volume 156 of *IFIP Conference Proceedings*, pages 225–240. Kluwer, 1999.

26 A. Petrenko, T. Higashino, and T. Kaji. Handling redundant and additional states in protocol testing. In A. Cavalli and S. Budkowski, editors, *Proceedings of the 8th International Workshop on Protocol Test Systems IWPTS '95,* Paris, France, pages 307–322, 1995.

27 Alexandre Petrenko and Nina Yevtushenko. Conformance tests as checking experiments for partial nondeterministic FSM. In Wolfgang Grieskamp and Carsten Weise, editors, *Formal Approaches to Software Testing, 5th International Workshop, FATES 2005, Edinburgh, UK, July 11, 2005, Revised Selected Papers*, volume 3997 of *Lecture Notes in Computer Science*, pages 118–133. Springer, 2005. doi:10.1007/11759744_9.

28 Alexandre Petrenko, Nina Yevtushenko, Alexandre Lebedev, and Anindya Das. Nondeterministic state machines in protocol conformance testing. In Omar Rafiq, editor, *Protocol Test Systems, VI, Proceedings of the IFIP TC6/WG6.1 Sixth International Workshop on Protocol Test systems, Pau, France, 28-30 September, 1993*, volume C-19 of *IFIP Transactions*, pages 363–378. North-Holland, 1993.

29 Robert Sachtleben. Unifying frameworks for complete test strategies. *Science of Computer Programming*, 237:103135, 2024. doi:10.1016/j.scico.2024.103135.

30 Adenilso Simão, Alexandre Petrenko, and Nina Yevtushenko. On reducing test length for FSMs with extra states. *Softw. Test. Verification Reliab.*, 22(6):435–454, 2012. doi:10.1002/STVR.452.

31 Wouter Smeenk, Joshua Moerman, Frits W. Vaandrager, and David N. Jansen. Applying automata learning to embedded control software. In Michael J. Butler, Sylvain Conchon, and Fatiha Zaïdi, editors, *Formal Methods and Software Engineering - 17th International Conference on Formal Engineering Methods, ICFEM 2015, France, 2015, Proceedings*, volume 9407 of *Lecture Notes in Computer Science*, pages 67–83. Springer, 2015. doi:10.1007/978-3-319-25423-4_5.

32 Michal Soucha and Kirill Bogdanov. SPYH-method: An improvement in testing of finite-state machines. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops, Västerås, Sweden, April 9-13, 2018*, pages 194–203. IEEE Computer Society, 2018. doi:10.1109/ICSTW.2018.00050.

33 A. S. Troelstra and H. Schwichtenberg. *Basic Proof Theory.* Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2 edition, 2000. doi:10.1017/CBO9781139168717.

34 Frits Vaandrager. Model learning. *Communications of the ACM*, 60(2):86–95, February 2017. doi:10.1145/2967606.

35 Frits Vaandrager. A new perspective on conformance testing based on apartness. In Venanzio Capretta, Robbert Krebbers, and Freek Wiedijk, editors, *Logics and Type Systems in Theory and Practice: Essays Dedicated to Herman Geuvers on The Occasion of His 60th Birthday*, pages 225–240, Cham, 2024. Springer Nature Switzerland. doi:10.1007/978-3-031-61716-4_15.

36 Frits W. Vaandrager, Paul Fiterau-Brostean, and Ivo Melse. Completeness of FSM test suites reconsidered. *CoRR*, abs/2410.19405, 2024. doi:10.48550/arXiv.2410.19405.

37 Frits W. Vaandrager, Bharat Garhewal, Jurriaan Rot, and Thorsten Wißmann. A new approach for active automata learning based on apartness. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*, volume 13243 of *Lecture Notes in Computer Science*, pages 223–243. Springer, 2022. doi:10.1007/978-3-030-99524-9_12.

38 Petra van den Bos, Ramon Janssen, and Joshua Moerman. n-Complete test suites for IOCO. *Softw. Qual. J.*, 27(2):563–588, 2019. doi:10.1007/S11219-018-9422-X.

39 M.P. Vasilevskii. Failure diagnosis of automata. *Cybernetics and System Analysis*, 9(4):653–665, 1973. (Translated from Kibernetika, No. 4, pp. 98-108, July-August, 1973.). doi:10.1007/BF01068590.

40 Nan Yang, Kousar Aslam, Ramon R. H. Schiffelers, Leonard Lensink, Dennis Hendriks, Loek Cleophas, and Alexander Serebrenik. Improving model inference in industry by combining active and passive learning. In Xinyu Wang, David Lo, and Emad Shihab, editors, *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019*, pages 253–263. IEEE, 2019. doi:10.1109/SANER.2019.8668007.

41 N. V. Yevtushenko and A. F. Petrenko. Synthesis of test experiments in some classes of automata. *Autom. Control Comput. Sci.*, 24(4):50–55, April 1991.

A Condition (1) is Needed

Proposition 4.13 establishes that condition (1) implies that all states in $F^{<k}$ are identified. The converse implication does not hold: even if all states in $B \cup F^{\leq k}$ are identified, condition (1) may not hold. The Mealy machines $\mathcal{S}$ and $\mathcal{M}$ of Figure 7 present a counterexample with $k = 1$ and $A = \{\epsilon, l, r\}$.

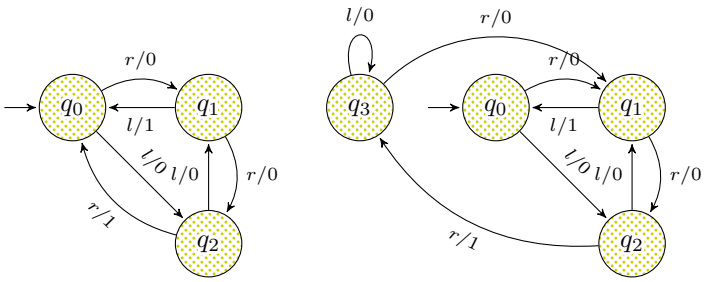


Figure 7 A specification $\mathcal{S}$ (left) and a faulty implementation $\mathcal{M}$ (right).

Note that these machines are not equivalent: input sequence *rrrlll* distinguishes them. The extra state q_3 of $\mathcal{M}$ behaves similar as state q_0 of $\mathcal{S}$, but is not equivalent. Figure 8 shows an observation tree $\mathcal{T}$ for both $\mathcal{S}$ and $\mathcal{M}$.

Observation tree $\mathcal{T}$ meets all the requirements of Theorem 4.9, except condition (1). One way to think of $\mathcal{T}$ is that $\mathcal{M}$ cherry picks distinguishing sequences from $\mathcal{S}$ to ensure that the F^1 states are identified by a sequence for which $\mathcal{S}$ and $\mathcal{M}$ agree. Note that B and F^0

are both complete, and all states in B and $F^{\leq 1}$ are identified. However, $\mathcal{T}$ does not satisfy condition (1) as t_{13} is not apart from t_6, $C(t_6) = \{t_0\}$ and $C(t_{13}) = \{t_2\}$. The example shows that without condition (1), Theorem 4.9 doesn't hold.

B The SPY and H-methods are not k-A-Complete

▶ **Example B.1.** Figure 9(left) shows the running example $\mathcal{S}$ from the article by Simão, Petrenko and Yevtushenko that introduces the SPY-method [30]. Using this method, a 3-complete test suite $\{aaaa, baababba, bbabaa\}$ was derived in [30]. Consider the minimal state cover $A = \{\epsilon, a\}$ for $\mathcal{S}$. Implementation $\mathcal{M}$ from Figure 9(right) is contained in fault domain $\mathcal{U}_1^A$, since all states can be reached via at most one transition from $0'$ and $1'$. Clearly $\mathcal{S} \not\approx \mathcal{M}$, as input sequence aab provides a counterexample. Nevertheless, $\mathcal{M}$ passes the derived test suite. Thus the test suite generated by the SPY-method [30] is not 1-A-complete.

▶ **Example B.2.** Figure 11 shows the tree for a 3-complete test suite generated by the H-method of Dorofeeva et al [6] for the machine of Figure 10(left). This tree satisfies condition (3) since the only transitions from F^0 to F^1 that change the candidate set are a-transitions, and the sources and targets of those transitions are apart. The machine of Figure 10(right) will pass this test suite, even though the two machines are inequivalent (cbc is a counterexample). It is easy to check that the machine on the right is in $\mathcal{U}_1^A$, for $A = \{\epsilon, a\}$. Thus the test suite generated by the H-method is not 1-A-complete. Indeed, the test suite does not meet condition (1) since (for example) the states with access sequences cb and ac have different candidate sets but are not apart.

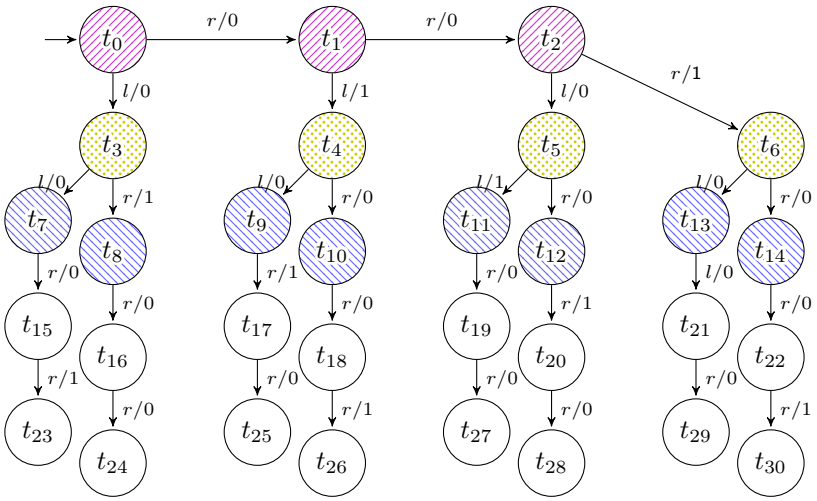


Figure 8 Observation tree for FSMs $\mathcal{S}$ and $\mathcal{M}$ from Figure 7.

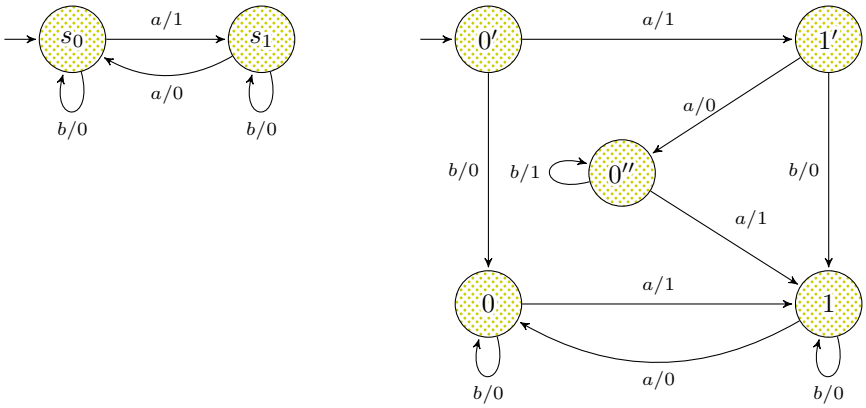


Figure 9 An implementation from fault domain $\mathcal{U}_1^A$ (right) that passes the 3-complete test suite $\{aaaa, baababba, bbabaa\}$ that was constructed for the specification (left) using the SPY-method.

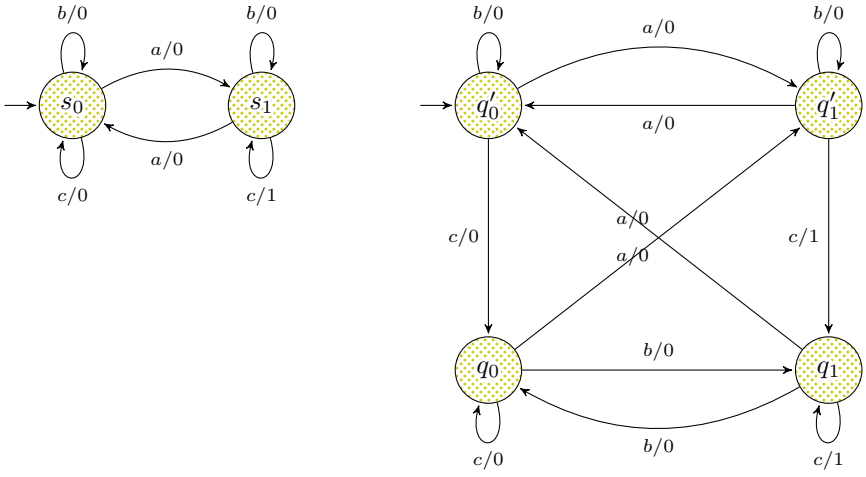


Figure 10 Specification (left) and implementation (right) from fault domain $\mathcal{U}_1^A$ that passes the test suite of Figure 11.

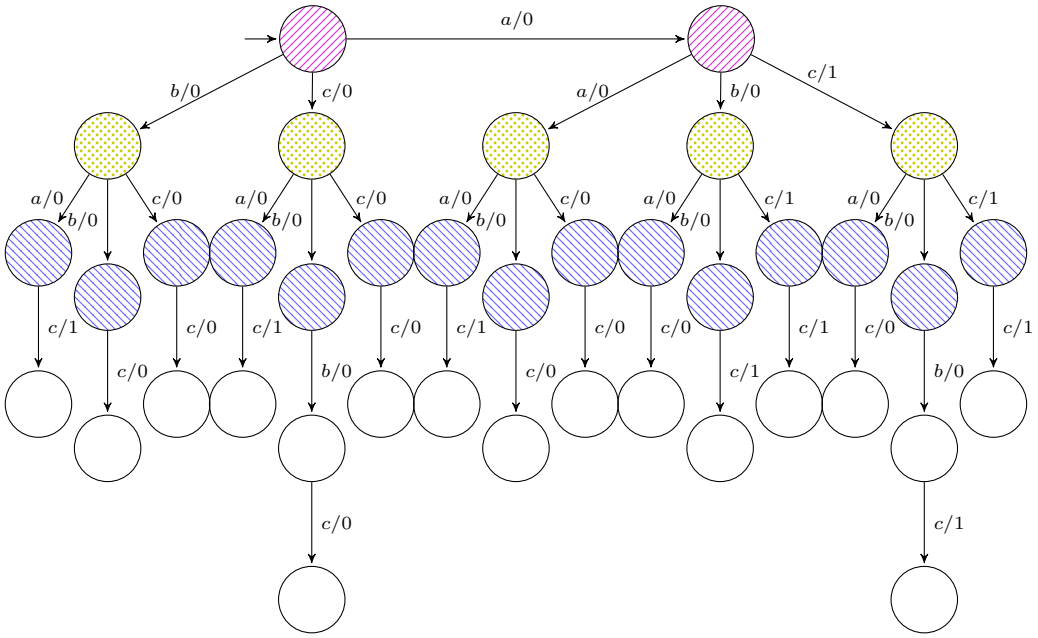


Figure 11 Testing tree for 3-complete test suite constructed for the specification of Figure 10(left) using the H-method.

From Bisimulation to Traces: The Impact of Parallel Composition on Finite Bases

Rowin Versteeg
Eindhoven University of Technology, The Netherlands

Valentina Castiglioni
Eindhoven University of Technology, The Netherlands

Bas Luttik
Eindhoven University of Technology, The Netherlands

Abstract

We consider process algebras with inaction, action prefix, non-deterministic choice and interleaving parallel composition modulo the behavioural equivalences in van Glabbeek's linear time-branching time spectrum, and study the existence of finite bases (i.e., finite sound and complete axiomatisations) for these algebras. We prove that if the alphabet of actions is infinite and the behavioural equivalence is either simulation equivalence or trace equivalence, then a finite basis exists and is obtained by extending the known ground-complete axiomatisations for these behavioural equivalences. We prove that if the alphabet of actions is finite, then a finite basis does not exist for these equivalences. We also prove for all behavioural equivalences between ready simulation and completed traces there cannot exist a finite basis irrespective of the cardinality of the alphabet of actions (provided that it is non-empty). Finally, we prove that these results are maintained if the process algebra is extended with a constant for successful termination.

2012 ACM Subject Classification Theory of computation → Equational logic and rewriting

Keywords and phrases Equational basis, Parallel composition, Preorders, Equivalences, Linear time - branching time spectrum

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2025.35

1 Introduction

Behavioural semantics have been introduced as simple and elegant tools for an explicit comparison of the semantics of processes. These are preorders and equivalence relations allowing one to establish whether two processes have the same *observable behaviour*: different notions of observability correspond to different levels of abstraction from the information on process execution, which can either be considered irrelevant in an application context, or be unavailable to an external observer. A taxonomy of behavioural relations based on their distinguishing power, known as the *linear time - branching time spectrum* (henceforth referred to as *the spectrum*), was proposed by van Glabbeek in [20].

Semantic properties of processes can also be defined, implicitly, by a set of *equational axioms*, i.e., syntactic equations over terms in a considered process algebra. Informally, if a term t is proved equal to a term u by means of the axioms, then we can say that t and u describe the same behaviour. An axiomatisation $\mathcal{E}$ is *ground-complete* for a behavioural semantics S if all variable-free terms (i.e., processes) that are related by S can be proved equal from $\mathcal{E}$. An axiomatisation $\mathcal{E}$ is *complete* for S if also terms with variables that are related by S can be proved equal from $\mathcal{E}$. We call a finite, sound and complete axiomatisation a *finite basis*. Obtaining an axiomatisation of a behavioural semantics is a key problem in concurrency theory, as it allows us to characterise the semantics of a process algebra in a purely syntactic fashion, making thus such a characterisation independent of the details of the definition of the behavioural semantics of interest. Moreover, an axiomatisation underlines

© Rowin Versteeg, Valentina Castiglioni, and Bas Luttik;
licensed under Creative Commons License CC-BY 4.0

Table 1 Existence of finite bases for the behavioural semantics over $\mathrm{BCCSP}_{\|}$. The rows are clustered to indicate the dependencies of the proofs.

	$\lvert A\rvert = 1$	$2 \leq \lvert A\rvert < \infty$	$\lvert A\rvert = \infty$
Bisimulation	-	-	-
2-Nested Simulation	?	-	-
Possible Futures	?	-	-
Ready Simulation	-	-	-
Possible Worlds	-	-	-
Ready Traces	-	-	-
Failure Traces	-	-	-
Readies	-	-	-
Failures	-	-	-
Completed Simulation	-	-	-
Completed Traces	-	-	-
Simulation	-	-	+
Traces	-	-	+

the differences between the various semantics via a collection of revealing axioms, and, due to its syntactic nature, it can be applied in verification tools based on theorem proving or rewriting.

As part of the analysis of the behavioural semantics in the spectrum, van Glabbeek proposed finite, ground-complete axiomatisations for them over the language BCCSP, which consists of the basic operators of CCS [26] and CSP [24], i.e., inaction, action prefix, and nondeterministic choice. However, the majority of process algebras include some form of *parallel composition operator* to model the concurrent interaction between processes. Hence, the paper [2] proposed a systematic study of the axiomatisability of the CCS interleaving parallel composition operator "$\|$" modulo the behavioural semantics in the spectrum. The study was carried out over the language $\mathrm{BCCSP}_{\|}$, i.e., BCCSP extended with $\|$. Specifically, it was shown that all behavioural semantics that are at least as coarse as ready simulation admit a finite ground-complete axiomatisation over $\mathrm{BCCSP}_{\|}$. Conversely, for all behavioural semantics that are at most as coarse as possible futures there exists no finite ground-complete axiomatisation, and, thus, no finite basis, provided the set of actions A over which the terms are built contains more than one element. (It is still an open problem to determine whether a finite ground-complete axiomatisation exists for possible futures and the n-nested trace and n-nested simulation semantics when $|A| = 1$.)

Our Contribution. In this paper, we move the focus from processes to terms with variables, and we investigate the existence of finite bases for the behavioural semantics in the spectrum over $\mathrm{BCCSP}_{\|}$. Table 1 reports an overview of our findings. We remark that the non-existence of finite bases for possible futures, nested simulations, and bisimulation semantics follows directly from the non-existence of finite ground-complete axiomatisations over $\mathrm{BCCSP}_{\|}$ for them [2, 7, 27]. Interestingly, for the other semantics, we show that almost all finite ground-completeness results from [2] turn into negative results. The only exception is constituted by trace and simulation equivalences over $\mathrm{BCCSP}_{\|}$ terms constructed over an infinite set of actions A. In this case, we provide, for each equivalence, an ω-complete axiomatisation that combined with the ground-completeness results from [2] give us the desired finite bases (Section 4.3). Here, by finite we mean *finitely representable*, namely, the axiomatisation

includes only finitely many axiom schemas (that can generate infinitely many equations). The ω-completeness result is obtained by making use of Groote's technique of inverted substitutions [21] and two additional axioms for $\|$ with respect to those in [2] (one of which is original with respect to the related literature). As soon as $|A| < \infty$, this positive results for trace and simulation semantics turn into negative ones. The interesting bit is that those negative results require different proof techniques, according to the cardinality of A. In the case $|A| = 1$ we use the transformation technique from [4] to establish a relation between $\mathrm{BCCSP}_{\|}$, modulo the two semantics, and the max-plus algebra over natural numbers [5]. We then use this relation and the non-finite axiomatisability result for the latter algebra [5] to conclude that the same negative result holds for the former algebra (Section 4.1). In the case $2 \leq |A| \leq \infty$, we prove the non-existence of a finite axiomatisation modulo trace and simulation semantics over $\mathrm{BCCSP}_{\|}$ by applying Moller's proof-theoretic technique [27] (Section 4.2). We make use of the same technique to prove the non-existence of finite basis modulo all semantics between ready simulation and completed traces over $\mathrm{BCCSP}_{\|}$, regardless of the cardinality of A (Section 3). Notably, we prove all negative results for both versions of each semantics: preorder and equivalence. Finally, we also discuss the lifting of our results to $\mathrm{BCCSP}_{\|}$ extended with successful termination (Section 5).

Due to space limitations, we present only a high-level description of our results and the proof techniques applied to obtain them. We refer the interested reader to [30] for the detailed technical development.

Related Work. The axiomatisations of BCCSP have been studied extensively, and we give here only a brief summary of the results. Van Glabbeek presented finite, sound and ground-complete axiomatisations for most equivalences in the spectrum [20]. However, some of these axiomatisations contained conditional axioms. In response Blom, Fokkink and Nain gave finite, sound and ground-complete axiomatisations for ready traces, ready simulation and failure traces without conditions [11]. They also proved that there exists no finite, sound and ground-complete axiomatisation for ready traces if the number of actions is infinite. Aceto et al. proved that there are no finite, sound and ground-complete axiomatisations for 2-nested-simulation and possible futures [7].

Moller proved the existence of a finite basis for bisimulation [27], Groote did this for completed traces and traces [21] and Fokkink and Nain for failures [19]. Conversely, Chen, Fokkink and Nain proved non-finite axiomatisability results for ready simulation and completed simulation [15], Chen and Fokkink for simulation [12], Fokkink and Nain for ready traces, readies and possible worlds [18] and Chen et al. for failure traces [13].

For what concerns preorders, Aceto, Fokkink and Ingólfsdóttir presented the transformation technique for BCCSP to transform finite, sound and (ground-)complete axiomatisations for a behavioural preorder to a (ground-)complete axiomatisation for the respective behavioural equivalence [6]. This method was applied only to the behavioural semantics between ready simulation and traces. The transformation technique can be used also to lift results across process algebras. Aceto, Ésik and Ingólfsdóttir proved there is no finite basis for the max-plus algebra on natural numbers, then used that fact to prove that trace equivalence with $|A| = 1$ has no finite basis over the process algebra BPA [4], which is similar to BCCSP.

Regarding $\mathrm{BCCSP}_{\|}$, Hennessy and Milner [22, 23], proposed a ground-complete axiomatisation over it modulo strong bisimilarity and weak bisimilarity. This axiomatisation contains the expansion law, an axiom schema which expresses equationally the semantics of the parallel composition operator. However, the expansion law generates an infinite number of axioms, which makes any axiomatisation containing it infinite. Moller then proved that no

Table 2 Operational semantics of $\mathrm{BCCSP}_{\|}$.

$$\frac{}{a.t \xrightarrow{a} t} \qquad \frac{t \xrightarrow{a} t'}{t + u \xrightarrow{a} t'} \qquad \frac{u \xrightarrow{a} u'}{t + u \xrightarrow{a} u'} \qquad \frac{t \xrightarrow{a} t'}{t \,\|\, u \xrightarrow{a} t' \,\|\, u} \qquad \frac{u \xrightarrow{a} u'}{t \,\|\, u \xrightarrow{a} t \,\|\, u'}$$

finite, sound and ground-complete axiomatisation exists for bisimulation [28]. As discussed above, Aceto et al. recently studied the existence of ground-complete axiomatisations for the other behavioural equivalences in the spectrum [2].

2 Preliminaries

The Process Algebra $\mathrm{BCCSP}_{\|}$. The language $\mathrm{BCCSP}_{\|}$ extends BCCSP with the interleaving parallel composition operator, and it is defined by the following grammar:

$$t ::= \mathbf{0} \mid x \mid a.t \mid t + t \mid t \,\|\, t$$

where a ranges over a non-empty set of actions A, and x ranges over a countably infinite set of variables V. We shall use the meta-variables $t, u, \ldots$ to range over the set $\mathcal{O}$ of *open* $\mathrm{BCCSP}_{\|}$ terms, and write $var(t)$ for the collection of variables occurring in the term t. We also adopt the standard convention that prefixing binds strongest and $+$ binds weakest. We use a *summation* $\sum_{i=1}^{n} t_i$ to denote the term $t = t_1 + \cdots + t_n$, where terms $t_1, \ldots, t_n$ do not have $+$ as head operator and are called *summands* of t. The empty sum represents $\mathbf{0}$. We let $t^n = t \,\|\, t^{n-1}$, with $t^1 = t$. A term is *closed* if it does not contain any variables. Closed terms are usually called *processes*, and we denote the set of $\mathrm{BCCSP}_{\|}$ closed terms by $\mathcal{P}$. A *substitution* $\sigma : V \to \mathcal{O}$ is a mapping from variables to terms. A *closed substitution* $\rho : V \to \mathcal{P}$ is a mapping from variables to closed terms.

We use the SOS framework [29] to equip processes with an operational semantics. A (*closed*) *literal* is an expression of the form $t \xrightarrow{a} t'$ for some (closed) terms t, t' and action $a \in A$. The inference rules for *prefixing* $a._$, *nondeterministic choice* $+$ and *interleaving parallel composition* $\|$ are reported in Table 2. These rules induce the *labelled transition system* (LTS) [25] $(\mathcal{P}, A, \to)$ whose transition relation $\to \subseteq \mathcal{P} \times A \times \mathcal{P}$ contains exactly the closed literals that can be derived using the rules in Table 2. As usual, we write $p \xrightarrow{a} p'$ in place of $(p, a, p') \in \to$, and $p \xrightarrow{a}$ if there exists some p' such that $p \xrightarrow{a} p'$. Let the set of *initial actions* of p be defined as $I(p) = \{a \in A \mid p \xrightarrow{a}\}$. Then, we write $p \to$ if $I(p) \neq \emptyset$, and $p \not\to$ otherwise. Given a sequence of actions $s = a_1 \cdots a_n \in A^*$, we write $p \xrightarrow{s} p'$ if and only if there exists a sequence of transitions $p = p_0 \xrightarrow{a_1} p_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} p_n = p'$. If $p \xrightarrow{s} p'$ holds for some process p', then s is a *trace* of p. Moreover, we say that s is a *completed trace* of p if $I(p') = \emptyset$. We let ε denote the empty trace, and $|s|$ denote the length of trace s. We use $s[i]$ to denote the action occurring at position $1 \leq i \leq |s|$. The prefix of s up to, and including, position i is denoted by $s[..i]$. Similarly, the suffix from position i onward is denoted by $s[i..]$. We write s^n for an n repetition of s, where $s^0 = \varepsilon$ and $s^n = ss^{n-1}$. For ease of notation, given $s \in A^*$ and a term t, we let $s.t = s[1].s[2..].t$, with $\varepsilon.t = t$.

Behavioural Semantics. To compare the behaviour of two processes, we use *behavioural semantics* (BS). These are preorders (reflexive and transitive relations) and equivalences over the states of a LTS allowing us to establish whether two processes have the same *observable* semantics. A *preorder* $p \precsim_{\mathsf{BS}} q$ specifies that the semantics of p is included in that of q,

according to the notion of observability specified by BS. We have that p and q are BS *equivalent*, denoted $p \simeq_{\mathsf{BS}} q$, if and only if $p \precsim_{\mathsf{BS}} q$ and $q \precsim_{\mathsf{BS}} p$. Van Glabbeek presented the *linear time–branching time spectrum* [20], henceforth *the spectrum*, which is a taxonomy of BS based on their distinguishing power. Henceforth, we write $\precsim$ (respectively, $\simeq$) to denote an arbitrary preorder (respectively, equivalence) in the spectrum. Below, we report the formal definitions of the BS that we will use in the technical development of our results.

▶ **Definition 1** (Trace and completed trace preorders). *We denote the set of traces of p by $\mathsf{T}(p) = \{s \in A^* \mid \exists p'.p \xrightarrow{s} p'\}$. Then $p \precsim_{\mathsf{T}} q$ if and only if $\mathsf{T}(p) \subseteq \mathsf{T}(q)$. Likewise, the set of completed traces of p is denoted by $\mathsf{CT}(p) = \{s \in A^* \mid \exists p'.p \xrightarrow{s} p' \wedge p' \not\rightarrow\}$. Then, $p \precsim_{\mathsf{CT}} q$ if and only if $\mathsf{T}(p) \subseteq \mathsf{T}(q)$ and $\mathsf{CT}(p) \subseteq \mathsf{CT}(q)$.*

We remark that, since all $\mathrm{BCCSP}_{\parallel}$ processes have traces of finite length, $\mathsf{CT}(p) \subseteq \mathsf{CT}(q)$ implies $\mathsf{T}(p) \subseteq \mathsf{T}(q)$ for all p, q. Hence, we have $p \precsim_{\mathsf{CT}} q$ if and only if $\mathsf{CT}(p) \subseteq \mathsf{CT}(q)$.

▶ **Definition 2** ((Bi)simulation relations). *A binary relation $\mathcal{R}$ over processes is a simulation if $p \mathcal{R} q$ and $p \xrightarrow{a} p'$ imply there is some q' such that $q \xrightarrow{a} q'$ and $p' \mathcal{R} q'$. Then $p \precsim_{\mathsf{S}} q$ if and only if there exists a simulation $\mathcal{R}$ such that $p \mathcal{R} q$.*
A simulation $\mathcal{R}$ is a ready simulation if $p \mathcal{R} q$ implies $I(p) = I(q)$. Then $p \precsim_{\mathsf{RS}} q$ if and only if there exists a ready simulation $\mathcal{R}$ such that $p \mathcal{R} q$.
A simulation $\mathcal{R}$ is a bisimulation if it is symmetric. Then $p \simeq_{\mathsf{B}} q$ if and only if there exists a bisimulation $\mathcal{R}$ such that $p \mathcal{R} q$.

Using literals and inference rules, we can consider transitions over terms containing variables, e.g. $a.x \xrightarrow{a} x$ and $x \not\rightarrow$, and extend the BS over terms as follows: for $t, u \in \mathcal{O}$, we let $t \precsim u$ if and only if $\rho(t) \precsim \rho(u)$ for all closed substitutions ρ. The following property is straightforward:

▶ **Lemma 3.** *Whenever $t \precsim u$, then $var(t) \subseteq var(u)$.*

Additionally, since $\mathrm{BCCSP}_{\parallel}$ operators are defined by inference rules in the de Simone format [17], all the preorders in the spectrum are precongruences with respect to them [8]. This means that $t_1 \precsim u_1$ and $t_2 \precsim u_2$ imply $a.t_1 \precsim a.u_1$ for any $a \in A$, $t_1 + t_2 \precsim u_1 + u_2$ and $t_1 \parallel t_2 \precsim u_1 \parallel u_2$. Likewise, any equivalence is a *congruence* over $\mathrm{BCCSP}_{\parallel}$ [8].

We define the *depth* of a term t, notation $|t|$, as the length of the longest trace in $\mathsf{CT}(t)$, formally defined as $|t| = \max\{|s| \mid s \in \mathsf{CT}(t)\}$. Notice that $|x| = 0$ for all $x \in V$. Then, for all behavioural preorders $\precsim$, the following property holds:

▶ **Lemma 4.** *For all terms t, u, if $t \precsim u$, then $|t| \leq |u|$.*

Moreover, a substitution cannot decrease the depth of a term.

▶ **Lemma 5.** *For all terms t and substitutions σ, $|t| \leq |\sigma(t)|$.*

Equational logic. In this paragraph we will use only equivalences to introduce notation and concepts. However, those can be extended to preorders. An *axiom system*, or *axiomatisation*, $\mathcal{E}$ is a collection of *equations* $t \approx u$ over the considered language, thus $\mathrm{BCCSP}_{\parallel}$ in this paper. An equation $t \approx u$ is *derivable* from an axiom system $\mathcal{E}$, notation $\mathcal{E} \vdash t \approx u$, if there is an equational proof for it from $\mathcal{E}$, namely if $t \approx u$ can be inferred from the axioms in $\mathcal{E}$ using the rules of equational logic. An axiom $t \approx u$ is *sound* modulo $\simeq$ if $t \simeq u$ holds. An axiomatisation $\mathcal{E}$ is sound modulo $\simeq$, if all of its axioms are sound. Consequently, if $\simeq$ is a congruence and $\mathcal{E}$ is sound modulo $\simeq$ then $\mathcal{E} \vdash t \approx u$ implies $t \simeq u$. Then, an axiom system

Table 3 The axiomatisation $\mathcal{E}_1$, which is sound modulo $\simeq_B$.

A0	$x + \mathbf{0} \approx x$	P0	$x \parallel \mathbf{0} \approx x$
A1	$x + y \approx y + x$	P1	$x \parallel y \approx y \parallel x$
A2	$(x + y) + z \approx x + (y + z)$		
A3	$x + x \approx x$		
EL1	$a.x \parallel b.y \approx a.(x \parallel b.y) + b.(a.x \parallel y)$		

$\mathcal{E}$ is *ground-complete* modulo $\simeq$ if $p \simeq q$ implies $\mathcal{E} \vdash p \approx q$ for all closed terms p and q. $\mathcal{E}$ is *ω-complete* modulo $\simeq$ if $\mathcal{E} \vdash \rho(t) \approx \rho(u)$ for all closed substitutions ρ implies $\mathcal{E} \vdash t \approx u$. Lastly, $\mathcal{E}$ is *complete* modulo $\simeq$ if $t \simeq u$ implies $\mathcal{E} \vdash t \approx u$ for all terms t and u.

▶ **Lemma 6.** *Any ground- and ω-complete axiomatisation modulo $\simeq$ is complete modulo $\simeq$.*

Hence, any complete axiomatisation is also ground-complete. An axiomatisation is *finite* if it contains a finite number of axioms. We say that there exists a *finite basis* modulo $\simeq$ if there exists a finite, sound and complete axiomatisation modulo $\simeq$. Consider, for instance axiomatisation $\mathcal{E}_1$ in Table 3. $\mathcal{E}_1$ is sound modulo $\simeq_{\mathsf{B}}$ [2], and, thus, modulo any equivalence in the spectrum. If an axiom contains an action, e.g. axiom EL1, then this may be instantiated with any action in A. Hence, if $|A| = \infty$, this axiom actually represents an infinite number of axioms. However, in this case we still consider the axiomatisation to be finite, since we interpret occurrences of actions in axioms as variables.

We present another axiom, P2, modelling associativity of interleaving:

P2 $\quad (x \parallel y) \parallel z \approx x \parallel (y \parallel z)$

We can show that P2 is sound modulo all the equivalences in the spectrum, and it will be used in the technical results for every finite basis presented in this work.

▶ **Lemma 7.** *Axiom* P2 *is sound modulo $\simeq_{\mathsf{B}}$.*

Henceforth, we write all terms modulo A0-A3 and P0-P2, to improve readability.

Breadth. We conclude this section by introducing the novel notion of *breadth* of a term, that is the largest nested parallel composition of variables in the term. Formally, we define the breadth of t, notation $[\![t]\!]$, inductively as follows:

$$[\![\mathbf{0}]\!] = 0 \qquad [\![x]\!] = 1 \qquad [\![a.t']\!] = [\![t']\!]$$
$$[\![t_1 + t_2]\!] = \max([\![t_1]\!], [\![t_2]\!]) \qquad [\![t_1 \parallel t_2]\!] = [\![t_1]\!] + [\![t_2]\!]$$

Notice that $[\![t]\!] = 0$ if and only if $var(t) = \emptyset$. Breadth satisfies the following property:

▶ **Lemma 8.** *For all terms t, u, if $t \precsim u$, then $[\![t]\!] \leq [\![u]\!]$.*

3 From Ready Simulation to Completed Traces

In this section we show that there exists no finite basis modulo any $\precsim_{\mathsf{RS}} \subseteq \precsim \subseteq \precsim_{\mathsf{CT}}$ or any $\simeq_{\mathsf{RS}} \subseteq \simeq \subseteq \simeq_{\mathsf{CT}}$, as soon as the set of actions A is not empty. To this end, we use the *proof-theoretic technique* [27]: This consists in providing a property of terms that is invariant under provability from finite axiom systems that are sound modulo $\simeq$ ($\precsim$). Then one can identify an infinite family of (in)equations, all sound modulo $\simeq$ ($\precsim$) in which the property is not preserved, i.e., one side of each (in)equation satisfies it, but the other side does not. This implies that there are infinitely many (in)equations that cannot be derived from any finite, sound, axiom system, and thus that there is no finite basis modulo $\simeq$ ($\precsim$).

3.1 Negative Results for the Preorders

Our main objective is to prove the following theorem.

▶ **Theorem 9.** *No precongruence* $\precsim$ *s.t.* $\precsim_{\mathsf{RS}} \subseteq \precsim \subseteq \precsim_{\mathsf{CT}}$ *admits a finite basis over* $BCCSP_{\|}$.

Following the strategy above, we introduce an infinite family of sound inequations for which there exist no finite, sound, axiomatisation from which we can derive all of them:

$$\{x \preceq x + x^n \mid n \geq 2\} \tag{1}$$

▶ **Lemma 10.** *The infinite family of inequations in* (1) *is sound modulo* $\precsim_{\mathsf{RS}}$.

Intuitively, if we have a closed substitution ρ then $\rho(x)$ is simulated by $\rho(x + x^n)$ and $I(\rho(x)) = I(\rho(x + x^n))$. Moreover, Lemma 10 implies that all the inequations in (1) are also sound modulo $\precsim$ for all $\precsim_{\mathsf{RS}} \subseteq \precsim \subseteq \precsim_{\mathsf{CT}}$.

Next, we identify a property of terms that is invariant under provability from a finite axiom system $\mathcal{E}$ that is sound modulo $\precsim_{\mathsf{CT}}$. To this end, notice that, since $\mathcal{E}$ is finite, it can only contain terms with a breadth less than some $n \geq 2$. Consider then a derivation $\mathcal{E} \vdash t \preceq u$ where $u \precsim_{\mathsf{CT}} x + x^n$ and $t \simeq_{\mathsf{CT}} x$. We will argue that no derivation step can increase the breadth of the term, showing that $u \simeq_{\mathsf{CT}} x$ must hold as well. The following proposition shows that this property is an invariant under provability from finite, sound, axiom systems.

▶ **Proposition 11.** *Let* $\mathcal{E}$ *be a finite set of inequations over* $\mathrm{BCCSP}_{\|}$ *that is sound modulo* $\precsim_{\mathsf{CT}}$. *Let* $n \geq 2$ *be greater than the breadth of any term in* $\mathcal{E}$. *Assume that:* $\mathcal{E} \vdash t \preceq u$; $u \precsim_{\mathsf{CT}} x + x^n$; *and* $t \simeq_{\mathsf{CT}} x$. *Then* $u \simeq_{\mathsf{CT}} x$.

The following properties are straightforward:

1. $t \precsim_{\mathsf{CT}} u$, by the soundness of $\mathcal{E}$;
2. $|t| = |u| = 0$, by Lemma 4;
3. $[\![t]\!] = 1$, by Lemma 8;
4. $var(t) = var(u) = \{x\}$, by Lemma 3.

The proof of Proposition 11 is by induction on the length of the derivation of $t \preceq u$ from $\mathcal{E}$, with a case distinction on the last rule used in the derivation. We will sketch here the proof of the most interesting case, i.e. the case of the substitution rule. The core of the proof lies in determining the syntactic structure of t and u. We do this by means of the novel notion of breadth: by showing that $[\![u]\!] = 1$ must hold, we can determine that $u \simeq_{\mathsf{CT}} x$. To this end, we present some properties on the depth and breadth of terms as in Proposition 11:

▶ **Lemma 12.** *Let* t, u *be* $BCCSP_{\|}$ *terms:*

1. *If* $t \precsim_{\mathsf{CT}} x + x^n$ *with* $n \geq 2$, *then* $[\![t]\!] = 1$ *or* $[\![t]\!] = n$.
2. *If* $t \precsim_{\mathsf{CT}} u$ *and* $|u| = 0$, *then* $var(u) \subseteq var(t)$.
3. *If* $|t| = [\![t]\!] = 0$, *then* $t \simeq_{\mathsf{CT}} \mathbf{0}$.
4. *If* $|t| = 0$, $[\![t]\!] = 1$ *and* $var(t) = \{x\}$, *then* $t \simeq_{\mathsf{CT}} x$.

Assume that $\mathcal{E} \vdash t \preceq u$ because $t = \sigma(v)$ and $u = \sigma(w)$ from some $v \preceq w \in \mathcal{E}$ and substitution σ. We have $\sigma(v) \precsim_{\mathsf{CT}} \sigma(w)$, and since $\sigma(w) \precsim_{\mathsf{CT}} x + x^n$, we can use Lemma 12.1 to obtain $[\![\sigma(w)]\!] = 1$ or $[\![\sigma(w)]\!] = n$. We argue that $[\![\sigma(w)]\!] = 1$ must hold, so assume, towards a contradiction, $[\![\sigma(w)]\!] = n$. We have $[\![w]\!] < n$, so there must be some variable $y \in var(w)$ such that $2 \leq [\![\sigma(y)]\!] \leq n$. By the soundness of $\mathcal{E}$ we have $v \precsim_{\mathsf{CT}} w$. Since $\sigma(w) \precsim_{\mathsf{CT}} x + x^n$, by Lemma 4 we get $|\sigma(w)| = 0$, from which, by Lemma 5, we infer $|w| = 0$. Hence, by Lemma 12.2 we obtain $y \in var(v)$. Then we reach a contradiction, since $[\![\sigma(v)]\!] \geq [\![\sigma(y)]\!] > 1$ contradicts $[\![\sigma(v)]\!] = 1$. Consequently, $[\![\sigma(w)]\!] = 1$, so $\sigma(w) \simeq_{\mathsf{CT}} x$ by Lemma 12.4.

Theorem 9 then follows by noticing that no inequation in (1) satisfies the considered property. In fact, while it is clearly the case that $x \simeq_{\mathsf{CT}} x$ and $x + x^n \precsim_{\mathsf{CT}} x + x^n$, it is not true that $x + x^n \simeq_{\mathsf{CT}} x$. In light of Proposition 11, this means that no finite axiomatisation that is sound modulo $\precsim_{\mathsf{CT}}$ can derive all the inequations in (1). Hence, there is not finite basis for $\precsim_{\mathsf{CT}}$ over $\mathrm{BCCSP}_{\|}$. Then, recall that the considered inequations are also sound modulo $\precsim_{\mathsf{RS}}$ (Lemma 10). Let $\precsim_{\mathsf{RS}} \subseteq \precsim \subseteq \precsim_{\mathsf{CT}}$ be a precongruence over $\mathrm{BCCSP}_{\|}$, and let $\mathcal{E}$ be any finite inequational axiomatisation for $\mathrm{BCCSP}_{\|}$ that is sound modulo $\precsim$. Since $\precsim \subseteq \precsim_{\mathsf{CT}}$, we have that $\mathcal{E}$ is sound modulo $\precsim_{\mathsf{CT}}$. Let $n \geq 2$ and larger than the breadth of each term in the inequations in $\mathcal{E}$. Then $x \preceq x + x^n$ cannot be derived from $\mathcal{E}$. Since this inequation is sound modulo $\precsim$, it follows that $\mathcal{E}$ is not complete modulo $\precsim$.

3.2 Negative Results for the Equivalences

We now show how we can extend Theorem 9 from preorders to equivalences. Specifically, the goal is to prove the following theorem.

▶ **Theorem 13.** *No congruence $\simeq$ s.t. $\simeq_{\mathsf{RS}} \subseteq \simeq \subseteq \simeq_{\mathsf{CT}}$ admits a finite basis over $BCCSP_{\|}$.*

From Lemma 10, we infer that the inequation $a.x \preceq a.(x + x^n)$ is sound modulo $\precsim_{\mathsf{RS}}$ for any $a \in A$. Then, we consider the infinite family of equations:

$$\{a.x + a.(x + x^n) \approx a.(x + x^n) \mid n \geq 2\}. \tag{2}$$

▶ **Lemma 14.** *The infinite family of equations in* (2) *is sound modulo $\simeq_{\mathsf{RS}}$.*

Intuitively, for any closed substitution ρ, $\rho(a.x + a.(x + x^n))$ is simulated by $\rho(a.(x + x^n))$ and vice versa. Additionally, the set of initial actions is $\{a\}$. As a direct consequence of Lemma 14, every equation in (2) is sound modulo $\simeq$ for all $\simeq_{\mathsf{RS}} \subseteq \simeq \subseteq \simeq_{\mathsf{CT}}$.

We can prove that the equations in (2) are not derivable from any finite axiom system $\mathcal{E}$ which is sound modulo $\simeq_{\mathsf{CT}}$ and contains terms up to breadth $n \geq 2$. Informally, consider the derivation $\mathcal{E} \vdash a.x + a.(x + x^n) \approx a.(x + x^n)$. The question is whether we can apply an axiom $v \approx w \in \mathcal{E}$, with $\sigma(v) = a.x + a.(x + x^n)$ and $\sigma(w) = a.(x + x^n)$. Since w has a breadth less than n, there must be some variable $y \in var(v)$ encapsulating some part of x^n. However, equation $a.x + a.(x + x^k \| y) \approx a.(x + x^k \| y)$ with $k \geq 1$ is not sound modulo $\simeq_{\mathsf{CT}}$. Unfortunately, the proof is not this immediate, since the direct application of the axiom does not cover the general case in the considered derivation. Still, notice that $a.x \xrightarrow{a} x$, while $a.(x + x^n) \xrightarrow{a} x + x^n$ is the only transition from $a.(x + x^n)$. Then, we show that having an a-derivative that is completed trace equivalent to a variable is an invariant under provability from finite, sound, axiom systems. Afterwards, we can argue that no equation in (2) satisfies this invariant, from which the non-existence of a finite basis modulo $\simeq_{\mathsf{CT}}$ follows.

▶ **Proposition 15.** *Let $\mathcal{E}$ be a finite set of equations over* $\mathrm{BCCSP}_{\|}$ *that is sound modulo $\simeq_{\mathsf{CT}}$. Let $n \geq 2$ be greater than the breadth of any term in $\mathcal{E}$. Assume that: $\mathcal{E} \vdash t \approx u$; $u \precsim_{\mathsf{CT}} a.(x + x^n)$; and $t \xrightarrow{a} t'$ for some $t' \simeq_{\mathsf{CT}} x$. Then $u \xrightarrow{a} u'$ for some $u' \simeq_{\mathsf{CT}} x$.*

Also in this case we have some straightforward properties:

1. $t \simeq_{\mathsf{CT}} u$ by the soundness of $\mathcal{E}$;
2. $|t| = |u| = 1$ by Lemma 4 and the fact that $t \xrightarrow{a}$;
3. $1 \leq [\![t]\!] = [\![u]\!] \leq n$ by Lemma 8 and the fact that $t' \simeq_{\mathsf{CT}} x$; and
4. $var(t) = var(u) = \{x\}$ by Lemma 3 and the fact that $t' \simeq_{\mathsf{CT}} x$.

Moreover, as we consider terms $t \precsim_{\mathsf{CT}} a.(x + x^n)$, the following lemma will be helpful.

▶ **Lemma 16.** *If $t \precsim_{\mathsf{CT}} a.w$ and $t \xrightarrow{a} t'$, then $t' \precsim_{\mathsf{CT}} w$.*

Proposition 15 is proved by induction on the derivation of $\mathcal{E} \vdash t \approx u$. We sketch only the substitution case. Assume that $\mathcal{E} \vdash t \approx u$ because $t = \sigma(v)$ and $u = \sigma(w)$ for some $v \approx w \in \mathcal{E}$ and substitution σ. We distinguish two cases, according to the derivation of $t \xrightarrow{a} t'$.

Assume first that there exists some v' such that $v \xrightarrow{a} v'$ and $\sigma(v') \simeq_{\mathsf{CT}} x$. Then consider the closed substitution ρ defined as $\rho(y) = \mathbf{0}$ if $[\![\sigma(y)]\!] \leq 1$ and $\rho(y) = a^2.\mathbf{0}$ otherwise. Then, $\rho(v') \not\to$, so $a \in \mathsf{CT}(\rho(v))$. From the soundness of $\mathcal{E}$, we have $v \simeq_{\mathsf{CT}} w$, so $a \in \mathsf{CT}(\rho(w))$. By construction of ρ, this gives that $w \xrightarrow{a} w'$ for some w' s.t. $\rho(w') \not\to$. Then, $\sigma(w) \xrightarrow{a} \sigma(w')$. By Lemma 16 we obtain $\sigma(w') \precsim_{\mathsf{CT}} x + x^n$, so by Lemma 1 it holds that either $[\![\sigma(w')]\!] = 1$ or $[\![\sigma(w')]\!] = n$. If $[\![\sigma(w')]\!] = 1$, then $\sigma(w') \simeq_{\mathsf{CT}} x$ by Lemma 4, so the invariant holds. If $[\![\sigma(w')]\!] = n$, then since $[\![w]\!] < n$, there exists some variable $z \in var(w')$ such that $2 \leq [\![\sigma(z)]\!] \leq n$. However, we reach a contradiction, since $\rho(w') \xrightarrow{a}$ contradicts $\rho(w') \not\to$.

Assume now that there is no v' such that $v \xrightarrow{a} v'$ and $\sigma(v') \simeq_{\mathsf{CT}} x$. We first prove that there is some $y \in var(v)$ s.t. $\sigma(y) \xrightarrow{a} t''$ and either $t' = t''$ or $t' = t''\|\sigma(v_{\|})$ for some $v_{\|}$. We have $\sigma(v) \xrightarrow{a} t'$, so there must exist some summand v' of v such that $\sigma(v') \xrightarrow{a} t'$. We proceed by a case distinction on the structure of v'. We consider only the case $v' = v_1\|v_2$. In this case, assume, without loss of generality, that $\sigma(v_1) \xrightarrow{a} t_1$ such that $t' = t_1\|\sigma(v_2)$. Note that $\sigma(v_2) \not\to$, since $|\sigma(v)| = 1$. We can then perform a case distinction on the summands of v_1 to prove that there is some $y \in var(v_1)$ such that $\sigma(y) \xrightarrow{a} t''$ and either $t' = t''\|\sigma(v_2)$ or $t' = t''\|\sigma(v'_{\|})\|\sigma(v_2) = t''\|\sigma(v'_{\|}\|v_2)$ for some $v'_{\|}$. If $t'' = t'$, it is clear that $t'' \simeq_{\mathsf{CT}} x$. If $t' = t''\|\sigma(v_{\|})$, from $t''\|\sigma(v_{\|}) \simeq_{\mathsf{CT}} x$ and Lemma 8 it follows that $[\![t''\|\sigma(v_{\|})]\!] = 1$. So either $[\![t'']\!] = 1$ and $[\![\sigma(v_{\|})]\!] = 0$ or vice versa. Then $t'' \simeq_{\mathsf{CT}} x$ and $\sigma(v_{\|}) \simeq_{\mathsf{CT}} \mathbf{0}$ or vice versa, by Lemmas 3, 4. Consider the closed substitution ρ' defined as $\rho'(y) = a^2.\mathbf{0}$, $\rho'(z) = \mathbf{0}$ if $[\![\sigma(z)]\!] \leq 1$ and $\rho'(z) = a^3.\mathbf{0}$ otherwise. Then $a^2 \in \mathsf{CT}(\rho'(v))$ in both forms of t', so, by soundness of $\mathcal{E}$, $a^2 \in \mathsf{CT}(\rho'(w))$. Hence, there is some summand w' of w such that $a^2 \in \mathsf{CT}(\rho'(w'))$. We expand only the case $w' = w_1\|w_2$. Assume, without loss of generality, that $a^2 \in \mathsf{CT}(\rho'(w_1))$ and $\mathsf{CT}(\rho'(w_2)) = \{\varepsilon\}$ (it is impossible that $a \in \mathsf{CT}(\rho'(w_1))$ and $a \in \mathsf{CT}(\rho'(w_2))$, since $|\sigma(w)| \leq 1$ and by construction of ρ'). From $a^2 \in \mathsf{CT}(\rho'(w_1))$, we can again do a case distinction on the summands of w_1. This process can be repeated a finite number of times, until the nested summand y is reached. Let $w_{\|}$ be the accumulation of parallel compositions, such that $\mathsf{CT}(\rho'(w_{\|})) = \{\varepsilon\}$. We obtain that $\sigma(w) \xrightarrow{a} t''\|\sigma(w_{\|})$. By Lemma 16 we must have that $t''\|\sigma(w_{\|}) \precsim_{\mathsf{CT}} x + x^n$. Hence, $[\![t''\|\sigma(w_{\|})]\!] = 1$ or $[\![t''\|\sigma(w_{\|})]\!] = n$ by Lemma 1. In the first case we obtain that $t''\|\sigma(w_{\|}) \simeq_{\mathsf{CT}} x$ by Lemma 4, so the invariant holds. In the second case we reason towards a contradiction as above. Hence, $a^2 \in \mathsf{CT}(\rho'(w))$ only holds if $\sigma(w) \xrightarrow{a} u'$ for some $u' \simeq_{\mathsf{CT}} x$.

As discussed above, the equations in (2) do not satisfy the invariant in Proposition 15. Hence Theorem 13 follows for $\simeq_{\mathsf{CT}}$. The same reasoning applied in Section 3.1 then allows us to conclude that Theorem 13 holds for any congruence included between $\simeq_{\mathsf{RS}}$ and $\simeq_{\mathsf{CT}}$.

4 Simulation and Traces

In this section we discuss the axiomatisability modulo trace and simulation semantics over $\mathrm{BCCSP}_{\|}$. Interestingly, we show that as long as the set of actions is finite, then no finite basis can be given. However, this changes when $|A| = \infty$ since we can provide complete axiomatisations including only finitely many axiom schemas characterising the considered semantics. Moreover, we need to apply various proof techniques to obtain these results. Specifically, if $|A| = 1$ we prove that there exist no finite bases modulo $\simeq_{\mathsf{S}}$ or $\simeq_{\mathsf{T}}$ using the transformation technique from [5]; whereas if $2 \leq |A| < \infty$ we prove that there exist no finite

bases modulo $\precsim_{\mathsf{S}}$, $\precsim_{\mathsf{T}}$, $\simeq_{\mathsf{S}}$ or $\simeq_{\mathsf{T}}$ using the proof-theoretic technique described in Section 3. Lastly, for $|A| = \infty$, we present finite bases modulo $\simeq_{\mathsf{S}}$ and $\simeq_{\mathsf{T}}$ using Groote's technique of inverted substitutions [21].

4.1 Negative Results with $|A| = 1$

The main result of this subsection is the following theorem.

▶ **Theorem 17.** *If $|A| = 1$, then no congruence $\simeq$ s.t. $\simeq_{\mathsf{S}} \subseteq \simeq \subseteq \simeq_{\mathsf{T}}$ admits a finite basis over $BCCSP_{\|}$.*

For this section, assume $A = \{a\}$ and that $\simeq$ represents a congruence between $\simeq_{\mathsf{S}}$ and $\simeq_{\mathsf{T}}$. To prove that the algebra of $\mathrm{BCCSP}_{\|}$ processes modulo $\simeq$ has no finite basis we apply the transformation technique: we show how a presupposed finite basis for the algebra of $\mathrm{BCCSP}_{\|}$ processes modulo $\simeq$ can be transformed into a finite basis for the so-called *max-plus algebra* of natural numbers with max and + as operations. Since it has been shown that the latter algebra is not finitely based [5], it then follows that the former cannot be finitely based either. A similar approach was applied by Aceto et al. to prove that there exists no finite basis modulo $\simeq_{\mathsf{T}}$ if $|A| = 1$ over the process algebra BPA [4].

The following lemma is straightforward to prove.

▶ **Lemma 18.** *If $|A| = 1$, then for all processes p, q, we have $p \simeq q$ if and only if $|p| = |q|$. Moreover, $|\mathbf{0}| = 0$, $|p + q| = \max(|p|, |q|)$ and $|p \parallel q| = |p| + |q|$.*

Note that Lemma 18 establishes an isomorphism from the algebra of closed $\mathrm{BCCSP}_{\|}$ expressions modulo $\simeq$ with constant $\mathbf{0}$ and binary operations + and $\|$ and the max-plus algebra with constant 0 and binary operations max and +. Hence, there is a correspondence between the equational theories of these two algebras that we shall now formalise. We need the following inductively defined mapping (with $\mathtt{W}$ the set of variables in the max-plus algebra):

$$\langle \mathbf{0} \rangle = 0 \qquad \langle t_1 + t_2 \rangle = \max(\langle t_1 \rangle, \langle t_2 \rangle)$$
$$\langle x \rangle = \mathtt{x}, \text{ where } \mathtt{x} \in \mathtt{W} \qquad \langle t_1 \parallel t_2 \rangle = \langle t_1 \rangle + \langle t_2 \rangle$$

The following lemma is then the counterpart of Lemma 18 on the max-plus algebra, since the mapping defined above combines the depth of terms with occurrences of variables.

▶ **Lemma 19.** *For all terms t, u without an action prefix, we have $t \simeq u$ if and only if $\langle t \rangle = \langle u \rangle$.*

Now suppose that $\mathcal{E}$ is a finite basis for the algebra $\mathrm{BCCSP}_{\|}$ processes modulo $\simeq$. Then, in particular, if $t \simeq u$, then $\mathcal{E} \vdash t \approx u$. Now $\mathcal{E}$ may include axioms with action prefixes, so we cannot readily transform all of $\mathcal{E}$ into a complete set of axioms for the max-plus algebra. The following lemma, however, establishes that whenever two (open) $\mathrm{BCCSP}_{\|}$ terms are equivalent modulo $\simeq$, then either both or neither have occurrences of the action prefix.

▶ **Lemma 20.** *If $t \simeq u$, then t contains an action prefix if and only if u contains an action prefix.*

Now, let $\mathcal{E}' \subseteq \mathcal{E}$ be obtained by removing all axioms with occurrences of the action prefix. By Lemma 20 we have, for all $\mathrm{BCCSP}_{\|}$-terms t and u without occurrences of action prefixes that $\mathcal{E} \vdash t \approx u$ if and only if $\mathcal{E}' \vdash t \approx u$. Denote by $\langle \mathcal{E}' \rangle$ the set of equations $\langle t \rangle \approx \langle u \rangle$ such that $(t \approx u) \in \mathcal{E}'$. Then, for t, u without occurrences of action prefixes, $\mathcal{E} \vdash t \approx u$ if and

only if $\langle \mathcal{E}' \rangle \vdash \langle t \rangle = \langle u \rangle$. It would follow that $\langle \mathcal{E}' \rangle$ is a finite basis for the max-plus algebra, which cannot exist. We conclude that the presupposed finite basis for the algebra of closed $\mathrm{BCCSP}_{\|}$ expressions modulo $\simeq$ can also not exist.

4.2 Negative Results with $2 \leq |A| < \infty$

We use the proof-theoretic technique discussed in Section 3 to prove the following:

▶ **Theorem 21.** *If $2 \leq |A| < \infty$, then no precongruence $\precsim$ s.t. $\precsim_{\mathsf{S}} \subseteq \precsim \subseteq \precsim_{\mathsf{T}}$ admits a finite basis over $BCCSP_{\|}$.*

▶ **Theorem 22.** *If $2 \leq |A| < \infty$, then no congruence $\simeq$ s.t. $\simeq_{\mathsf{S}} \subseteq \simeq \subseteq \simeq_{\mathsf{T}}$ admits a finite basis over $BCCSP_{\|}$.*

Throughout this section consider $A = \{a_1, \ldots, a_{|A|}\}$, where $a = a_1, b = a_2$ and $s = a_3 \cdots a_{|A|}$. Note that $s = \varepsilon$ if $|A| = 2$. Let $n \geq 1$, then consider the term

$$h_n = ab^n s.\mathbf{0} + b^n s.x \parallel x + a.(s.x \parallel x^n) + \sum_{i=1}^{|s|} ab^n s[..i-1].(s[i+1..].x \parallel x).$$

Then we introduce the following infinite family of inequations:

$$\{ab^n s.x \preceq h_n \mid n \geq 1\}. \tag{3}$$

Intuitively, for any closed substitution ρ we see that $\rho(ab^n s.x)$ is simulated by one of the summands of $\rho(h_n)$ because of the added parallel composition with x. If $\rho(x) \not\to$ then it is the first summand, and if $\rho(x) \xrightarrow{a}$, then it is the second summand, etc. Thus, for each of the $|A| + 1$ options there is a corresponding summand.

▶ **Lemma 23.** *The infinite family of inequations in* (3) *is sound modulo $\precsim_{\mathsf{S}}$.*

We argue that not all inequations in the family are derivable using a finite collection of inequations $\mathcal{E}$, which is sound modulo $\precsim_{\mathsf{T}}$ and contains terms up to depth $n \geq 1$. First, we give some intuition why this is the case. Consider term $ab^n s.x$, for which we know that $ab^n s.x \xrightarrow{ab^n s} x$. However, h_n cannot perform trace $ab^n s$ and reach a term with an occurrence of x. We introduce a new notion, named the front of a term, to formally express this.

▶ **Definition 24.** *Let $\mathsf{front}(t) \subseteq var(t)$ be defined inductively as follows:*

$$\mathsf{front}(\mathbf{0}) = \mathsf{front}(a.t') = \emptyset \qquad \mathsf{front}(x) = \{x\}$$
$$\mathsf{front}(t_1 + t_2) = \mathsf{front}(t_1 \parallel t_2) = \mathsf{front}(t_1) \cup \mathsf{front}(t_2)$$

Hence, $\mathsf{front}(t)$ is the set of all variables of t that do not occur after an action prefix in t.

This means that whenever we have a substitution σ such that $\sigma(x) \xrightarrow{a}$, then $x \in \mathsf{front}(t)$ implies $\sigma(t) \xrightarrow{a}$. Additionally, it follows that if $|t| = 0$, then $var(t) = \mathsf{front}(t)$. Using this definition we obtain the following proposition.

▶ **Proposition 25.** *Let $2 \leq |A| < \infty$ and let $\mathcal{E}$ be a finite collection of inequations over $\mathrm{BCCSP}_{\|}$ that is sound modulo $\precsim_{\mathsf{T}}$. Let $n \geq 1$ be greater than the depth of any term in $\mathcal{E}$. Assume that: $\mathcal{E} \vdash t \preceq u$; $u \precsim_{\mathsf{T}} h_n$; and $t \xrightarrow{ab^n s} t'$ for some t' with $x \in \mathsf{front}(t')$. Then $u \xrightarrow{ab^n s} u'$ for some u' with $x \in \mathsf{front}(u')$.*

Continuing our intuitive argument, consider $t = ab^n s.x$ and derivation $\mathcal{E} \vdash ab^n s.x \preceq u$ where $u \precsim_{\mathsf{T}} h_n$. Either we directly apply an axiom $v \preceq w \in \mathcal{E}$ on $ab^n s.x$, so $\sigma(v) = ab^n s.x$ and $\sigma(w) = u$, or $\mathcal{E} \vdash b^n s.x \preceq u'$ and $u = a.u'$. To proceed, we need a property of front:

▶ **Lemma 26.** *If $|A| > 2$, $a \in A$, $t \precsim_{\mathsf{T}} u$ and $t \xrightarrow{s'} t'$ for some t' with $x \in \mathsf{front}(t')$ and s' does not contain a, then $u \xrightarrow{s'} u'$ for some u' with $x \in \mathsf{front}(u')$.*

If $\mathcal{E} \vdash b^n s.x \preceq u'$ and $u = a.u'$, we can infer, from $b^n s.x \precsim_{\mathsf{T}} u'$ and Lemma 26, that $u' \xrightarrow{b^n s} u''$ and $x \in \mathsf{front}(u'')$, so that $a.u' \xrightarrow{ab^n s} u''$ and $x \in \mathsf{front}(u'')$. If axiom $v \preceq w \in \mathcal{E}$ is applied such that $\sigma(v) = ab^n s.x$ and $\sigma(w) = u$, then some part of $ab^n s.x$ must be encapsulated in a variable $y \in var(v)$, i.e. $v = (ab^n s)[..i].y$ for some $0 \leq i < n$. To see this, let us first introduce the notion of $\mathbf{0}$-nested summand.

▶ **Definition 27.** *t' is a $\mathbf{0}$-nested-summand of t, denoted $t' \sqsubseteq_{\mathbf{0}} t$, if and only if either:*
1. *t' is a summand of t, or*
2. *t has a summand $t_1 \parallel t_2$, where $t' \sqsubseteq_{\mathbf{0}} t_1$ and $t_2 \simeq_{\mathsf{T}} \mathbf{0}$, or*
3. *t has a summand $t_1 \parallel t_2$, where $t' \sqsubseteq_{\mathbf{0}} t_2$ and $t_1 \simeq_{\mathsf{T}} \mathbf{0}$.*

Using this new notion, we can prove the following lemma.

▶ **Lemma 28.** *If $|A| \geq 2$, $t \precsim_{\mathsf{T}} h_n$ and $t \xrightarrow{(ab^n s)[..i]} t'$ for some t' which has a $\mathbf{0}$-nested-summand of the form $t_1 \parallel t_2$ such that $t_1 \parallel t_2 \xrightarrow{(ab^n s)[i+1..]} t''$ for some t'' with $0 \leq i < n$ and $x \in \mathsf{front}(t'')$, then either $t_1 \simeq_{\mathsf{T}} \mathbf{0}$ or $t_2 \simeq_{\mathsf{T}} \mathbf{0}$.*

Hence, even if v was of the form $v_1 \parallel v_2$, thanks to Lemma 28 we can always eliminate one of the two parallel components, and repeat the reasoning on the remaining component until we reach a term $v' \sqsubseteq_{\mathbf{0}} v$ such that $y \in \mathsf{front}(v')$. Consequently, y is in the front of v after performing trace $(ab^n s)[..i]$ and x is in the front of $\sigma(y)$ after performing the remaining part of the trace, namely $(ab^n s)[i+1..]$. If $|A| \geq 3$, then we can infer from $v \precsim_{\mathsf{T}} w$ and Lemma 26 that y is in the front after performing the same trace in w. Hence, $\sigma(w) \xrightarrow{ab^n s} u'$ for some u' with $x \in \mathsf{front}(u')$. To complete our reasoning, we need a special case of Lemma 26 if $|A| = 2$, covering the case in which variable y is in the front of v after performing trace ab^k for some $k < n$. Note that the following lemma holds for $|A| > 2$ due to Lemma 26.

▶ **Lemma 29.** *If $|A| \geq 2$, $t \precsim_{\mathsf{T}} u$, $1 \leq k < |t|$ and $t \xrightarrow{ab^k} t'$ for some t' with $x \in \mathsf{front}(t')$, then there exist u' and $0 \leq \ell < |u|$ such that $u \xrightarrow{ab^\ell} u'$ and $x \in \mathsf{front}(u')$.*

From Lemma 29 we know that w can perform trace $ab^n[..k]$ for some $0 \leq k < |w|$ for which $y \in \mathsf{front}(w)$. We argue that $k = i$ must hold, because of the constraints of $\sigma(w)$ in regards to the depth and soundness.

Hence, we can conclude that, in all cases, $u = \sigma(w) \xrightarrow{ab^n} u'$ for some term u' with $x \in \mathsf{front}(u')$.

Theorem 21 then follows by applying the same reasoning used to conclude Section 3.

The infinite family of inequations in (3) results in an infinite family of equations as:

$$\{ab^n s.x + h_n \approx h_n \mid n \geq 1\}. \tag{4}$$

▶ **Lemma 30.** *The infinite family of equations in (4) is sound modulo $\simeq_{\mathsf{S}}$.*

Then, following a similar reasoning as used for Proposition 25, we can prove the invariance under equational derivation also in the case of equivalences.

Table 4 Additional axioms for $\simeq_{\mathsf{T}}$ and $\simeq_{\mathsf{S}}$.

SP1	$(x+y) \parallel (z+w) \approx x \parallel (z+w) + y \parallel (z+w) + (x+y) \parallel z + (x+y) \parallel w$		
TP	$(x+y) \parallel z \approx x \parallel z + y \parallel z$	SP2	$a.x \parallel (y+z) \approx a.(x \parallel (y+z)) + a.x \parallel y + a.x \parallel z$
T	$a.x + a.y \approx a.(x+y)$	S	$a.(x+y) \approx a.(x+y) + a.x$

▶ **Proposition 31.** *Let $2 \leq |A| < \infty$ and let $\mathcal{E}$ be a finite collection of equations over* $\mathrm{BCCSP}_{\parallel}$ *that is sound modulo $\simeq_{\mathsf{T}}$. Let $n \geq 1$ be greater than the depth of any term in $\mathcal{E}$. Assume that: $\mathcal{E} \vdash t \approx u$; $u \precsim_{\mathsf{T}} h_n$; and $t \xrightarrow{ab^n s} t'$ for some t' with $x \in \mathsf{front}(t')$. Then $u \xrightarrow{ab^n s} u'$ for some u' with $x \in \mathsf{front}(u')$.*

Similar arguments to those used to prove Theorem 21 allow us to prove Theorem 22.

4.3 Finite Bases with $|A| = \infty$

In [2] it was proved that the axiomatisation $\mathcal{E}_{\mathsf{T}} = \mathcal{E}_1 \cup \{\mathrm{T}, \mathrm{TP}\}$, with $\mathcal{E}_1$ given in Table 3 and T, TP from Table 4, is a finite, sound and ground-complete axiomatisation of $\simeq_{\mathsf{T}}$ over $\mathrm{BCCSP}_{\parallel}$. Similarly, the axiomatisation $\mathcal{E}_{\mathsf{S}} = \mathcal{E}_1 \cup \{\mathrm{S}, \mathrm{SP1}, \mathrm{SP2}\}$ is finite, sound and ground-complete modulo $\simeq_{\mathsf{S}}$, with S, SP1 and SP2 given in Table 4. In this section, we extend $\mathcal{E}_{\mathsf{T}}$ and $\mathcal{E}_{\mathsf{S}}$ with a finite number of sound axioms that make them ω-complete besides ground-complete, so that the existence of finite bases for the two equivalences follows from Lemma 6.

To this end, consider the following axiom, that, to the best of our knowledge, is original with respect to the related literature:

P3 $\quad a.x \parallel y \approx a.x \parallel y + a.(x \parallel y).$

▶ **Lemma 32.** *Axiom* P3 *is sound modulo $\simeq_B$.*

Consider $\mathcal{E}'_{\mathsf{T}} = \mathcal{E}_{\mathsf{T}} \cup \{\mathrm{P2}, \mathrm{P3}\}$. We use Groote's technique of inverted substitutions to prove that the axiomatisation $\mathcal{E}'_{\mathsf{T}}$ is ω-complete if the number of actions is infinite. In [21], it is proved that an axiomatization $\mathcal{E}$ is ω-complete if for each equation $t \approx u$, of which all closed instances can be derived from $\mathcal{E}$, we can define a mapping $R : \mathcal{P} \to \mathcal{O}$ such that:

1. We can define a closed substitution $\tilde{\rho}$ such that $\mathcal{E} \vdash R(\tilde{\rho}(t)) \approx t$ and $\mathcal{E} \vdash R(\tilde{\rho}(u)) \approx u$.
2. From $\mathcal{E}$ and $\{p_1 \approx q_1, p_2 \approx q_2, R(p_1) \approx R(q_1), R(p_2) \approx R(q_2)\}$ we can derive $R(a.p_1) \approx R(a.q_1)$ for all $a \in A$, $R(p_1 + p_2) \approx R(q_1 + q_2)$ and $R(p_1 \| p_2) \approx R(q_1 \| q_2)$.
3. For each $v \approx w \in \mathcal{E}$ and each closed substitution ρ, we have $\mathcal{E} \vdash R(\rho(v)) \approx R(\rho(w))$.

In our setting, given an equation $t \approx u$ such that $\mathcal{E}'_{\mathsf{T}} \vdash \rho(t) \approx \rho(u)$ for all closed substitutions ρ, we consider the following mapping $R : \mathcal{P} \to \mathcal{O}$, where a_x is a unique action for each $x \in V$ and a_x does not appear in either t or u:

$$R(\mathbf{0}) = \mathbf{0} \qquad R(a.p) = a.R(p) \text{ where } a \neq a_x \text{ for all } x \in V$$
$$R(a_x.p) = x \parallel R(p) \qquad R(p+q) = R(p) + R(q) \qquad R(p \parallel q) = R(p) \parallel R(q)$$

We remark that existence, and uniqueness, of action a_x is guaranteed by the fact that we are considering a countably infinite set of actions. Then, we can show, by induction on the structure of terms, that $\mathcal{E}'_{\mathsf{T}} \vdash R(\tilde{\rho}(t)) \approx t$ and $\mathcal{E}'_{\mathsf{T}} \vdash R(\tilde{\rho}(u)) \approx u$ for the closed substitution $\tilde{\rho}$ defined by $\tilde{\rho}(x) = a_x.\mathbf{0}$ for all $x \in V$. The other two properties follow by a case analysis on terms and axioms, respectively. In particular, we remark that the new axiom P3 is

fundamental in proving the third property for axiom EL1 (Table 3). In fact, we have that if $a \neq a_x$ for all $x \in V$ and $b = b_y$ for some $y \in V$, then, letting $\rho(x) = p$ and $\rho(y) = q$:

$$\begin{aligned}
&R(a.p \parallel b.q) \\
&= a.R(p) \parallel R(b.q) \overset{\text{P3}}{\approx} a.R(p) \parallel R(b.q) + a.(R(p) \parallel R(b.q)) \\
&= R(a.p) \parallel (y \parallel R(q)) + a.(R(p) \parallel R(b.q)) \overset{\text{P1,P2}}{\approx} y \parallel (R(a.p) \parallel R(q)) + a.(R(p) \parallel R(b.q)) \\
&\overset{\text{A1}}{\approx} a.(R(p) \parallel R(b.q)) + y \parallel (R(a.p) \parallel R(q)) = R(a.(p \parallel b.q) + b.(a.p \parallel q)).
\end{aligned}$$

The following Theorem is then a direct consequence of [21, Theorem 3.1].

▶ **Theorem 33.** *If $|A| = \infty$, then the axiomatisation $\mathcal{E}'_{\mathsf{T}}$ is ω-complete modulo $\simeq_{\mathsf{T}}$.*

We can then use the same technique, with the same mapping R and closed substitution $\tilde{\rho}$, to show that $\mathcal{E}'_{\mathsf{S}} = \mathcal{E}_{\mathsf{S}} \cup \{\text{P2}, \text{P3}\}$ is ω-complete if the number of actions is infinite.

▶ **Theorem 34.** *If $|A| = \infty$, then the axiomatisation $\mathcal{E}'_{\mathsf{S}}$ is ω-complete modulo $\simeq_{\mathsf{S}}$.*

We remark that axioms T, S, SP2 and P3 are axiom schemas, i.e., they generate an equation for every instantiation of the action prefix. This means that, when $|A| = \infty$, they generate infinitely many axioms. However, as stated in the previous sections, we still consider the axiomatisations $\mathcal{E}_{\mathsf{T}'}$ and $\mathcal{E}_{\mathsf{S}'}$ as finite, since they are finitely representable (they include only finitely many axiom schemas).

5 Extension with Successful Termination

In this section we extend $\text{BCCSP}_{\parallel}$ with successful termination, and we discuss which results from the previous sections can be lifted to the new language.

First of all, we extend the definition of a LTS with successful termination, i.e., a LTS is a tuple $(\mathcal{P}, A, \rightarrow, \downarrow)$ where $\downarrow \subseteq \mathcal{P}$ is a set of successfully terminating processes. We write $p\downarrow$ if and only if $p \in \downarrow$. We extend the initial actions of a process p by successful termination ($\downarrow$), namely we define $\mathsf{menu}(p) = I(p) \cup \{\downarrow\}$ if $p\downarrow$ and $\mathsf{menu}(p) = I(p)$ otherwise. Then we extend the behavioural semantics with requirements on successful termination.

▶ **Definition 35** (Trace and completed trace preorders with termination)**.** *The set of terminating traces of p is denoted by $\mathsf{T}^{\downarrow}(p) = \{s \in A^* \mid \exists p'.p \xrightarrow{s} p' \wedge p'\downarrow\}$. Then $p \precsim^{\downarrow}_{\mathsf{T}} q$ if and only if $\mathsf{T}(p) \subseteq \mathsf{T}(q)$ and $\mathsf{T}^{\downarrow}(p) \subseteq \mathsf{T}^{\downarrow}(q)$. Similarly, $p \precsim^{\downarrow}_{\mathsf{CT}} q$ if and only if $\mathsf{T}(p) \subseteq \mathsf{T}(q)$, $\mathsf{CT}(p) \subseteq \mathsf{CT}(q)$ and $\mathsf{T}^{\downarrow}(p) \subseteq \mathsf{T}^{\downarrow}(q)$.*

▶ **Definition 36** ((Bi)simulations with termination)**.** *A simulation $\mathcal{R}$ is terminating if $p\mathcal{R}q$ and $p\downarrow$ imply $q\downarrow$. Then $p \precsim^{\downarrow}_{\mathsf{S}} q$ if and only if there exists a terminating simulation $\mathcal{R}$ such that $p\mathcal{R}q$.*
A terminating simulation $\mathcal{R}$ is a terminating ready simulation if $p\mathcal{R}q$ implies $\mathsf{menu}(p) = \mathsf{menu}(q)$. Then $p \precsim^{\downarrow}_{\mathsf{RS}} q$ if and only if there exists a terminating ready simulation $\mathcal{R}$ such that $p\mathcal{R}q$.
A terminating simulation $\mathcal{R}$ is a terminating bisimulation relation if it is symmetric. Then $p \simeq^{\downarrow}_{\mathsf{B}} q$ if and only if there exists a terminating bisimulation relation $\mathcal{R}$ such that $p\mathcal{R}q$.

We then extend $\text{BCCSP}_{\parallel}$ with the constant process $\mathbf{1}$, denoting successful termination, obtaining the language $\text{BSP}_{\parallel}$. The operational semantics of $\text{BSP}_{\parallel}$ is obtained by adding the inference rules in Table 5 to those in Table 2. While most axioms of $\mathcal{E}_1$ remain sound, P0 does not. In Table 6, we introduce the axioms P0a, P0b and P0c to replace P0.

▶ **Lemma 37.** *Axiom system $\mathcal{E}_2$, consisting of the axioms in Table 6, is sound modulo $\simeq^{\downarrow}_{\mathsf{B}}$.*

Table 5 Operational semantics of successful termination in $\mathrm{BSP}_{\|}$.

$$\frac{}{\mathbf{1}\downarrow} \qquad \frac{t\downarrow}{t+u\downarrow} \qquad \frac{u\downarrow}{t+u\downarrow} \qquad \frac{t\downarrow \quad u\downarrow}{t\,\|\,u\downarrow}$$

Table 6 The axiomatisation $\mathcal{E}_2$, which is sound modulo $\simeq_{\mathsf{B}}^{\downarrow}$.

A0	$x+\mathbf{0}\approx x$	P0a	$\mathbf{0}\,\|\,\mathbf{0}\approx\mathbf{0}$
A1	$x+y\approx y+x$	P0b	$x\,\|\,\mathbf{1}\approx x$
A2	$(x+y)+z\approx x+(y+z)$	P0c	$a.x\,\|\,\mathbf{0}\approx a.(x\,\|\,\mathbf{0})$
A3	$x+x\approx x$	P1	$x\,\|\,y\approx y\,\|\,x$
EL1	$a.x\,\|\,b.y\approx a.(x\,\|\,b.y)+b.(a.x\,\|\,y)$		

5.1 Preservation of Negative Results

We present a schema to apply the transformation technique on any finite basis for $\mathrm{BSP}_{\|}$ into a finite basis for $\mathrm{BCCSP}_{\|}$. This schema also works for finite, sound and ground-complete axiomatisations. Consequently, for every behavioural semantics it holds that the non-existence of a finite basis for $\mathrm{BCCSP}_{\|}$ implies the non-existence of a finite basis for $\mathrm{BSP}_{\|}$.

Consider some behavioural semantics in the spectrum, let $\precsim$ and $\precsim^{\downarrow}$ be the corresponding preorders and let $\simeq$ and $\simeq^{\downarrow}$ be the corresponding equivalences. We argue that if $t\precsim u$ holds, then $t\,\|\,\mathbf{0}\precsim^{\downarrow}u\,\|\,\mathbf{0}$ must hold. Since from the operational semantics it is clear that for any closed $\mathrm{BSP}_{\|}$ substitution ρ no successfully terminating term can be reached from $\rho(t\,\|\,\mathbf{0})$ or $\rho(u\,\|\,\mathbf{0})$, then any closed substitution which uses the successful termination constant has no influence on the behaviour. We define $[t]_{\mathbf{1}\mapsto\mathbf{0}}$ to be $\mathrm{BSP}_{\|}$ term t with every occurrence of the constant $\mathbf{1}$ replaced with the constant $\mathbf{0}$. We then also argue that $t\precsim^{\downarrow}u$ implies $[t]_{\mathbf{1}\mapsto\mathbf{0}}\precsim[u]_{\mathbf{1}\mapsto\mathbf{0}}$, since the successful termination constant has no influence on the old criteria. Using these arguments, consider a finite basis modulo $\simeq^{\downarrow}$, let this be $\mathcal{E}$. We argue that $[\mathcal{E}]_{\mathbf{1}\mapsto\mathbf{0}}\cup\{\mathrm{P0}\}$, where $[\mathcal{E}]_{\mathbf{1}\mapsto\mathbf{0}}=\{[v]_{\mathbf{1}\mapsto\mathbf{0}}\approx[w]_{\mathbf{1}\mapsto\mathbf{0}}\mid v\approx w\in\mathcal{E}\}$, then serves as a finite basis modulo $\simeq$, leading to a contradiction with Theorem 13. Therefore, we can conclude that the following theorem holds.

▶ **Theorem 38.** *No congruence $\simeq^{\downarrow}$ s.t. $\simeq_{\mathsf{RS}}^{\downarrow}\subseteq\simeq^{\downarrow}\subseteq\simeq_{\mathsf{CT}}^{\downarrow}$ admits a finite basis over $\mathrm{BSP}_{\|}$. If $|A|<\infty$, then no congruence $\simeq^{\downarrow}$ s.t. $\simeq_{\mathsf{S}}^{\downarrow}\subseteq\simeq^{\downarrow}\subseteq\simeq_{\mathsf{T}}^{\downarrow}$ admits a finite basis over $\mathrm{BSP}_{\|}$.*

We can use the same schema also for the preorders, provided we substitute axiom P0 with the inequations P0-l ($x\,\|\,\mathbf{0}\preceq x$) and P0-r ($x\preceq x\,\|\,\mathbf{0}$), which are sound modulo $\simeq_{\mathsf{B}}$.

▶ **Theorem 39.** *No precongruence $\precsim^{\downarrow}$ s.t. $\precsim_{\mathsf{RS}}^{\downarrow}\subseteq\precsim^{\downarrow}\subseteq\precsim_{\mathsf{CT}}^{\downarrow}$ admits a finite basis over $\mathrm{BSP}_{\|}$. If $|A|<\infty$, then no precongruence $\precsim^{\downarrow}$ s.t. $\precsim_{\mathsf{S}}^{\downarrow}\subseteq\precsim^{\downarrow}\subseteq\precsim_{\mathsf{T}}^{\downarrow}$ admits a finite basis over $\mathrm{BSP}_{\|}$.*

5.2 Finite Basis with $|A|=\infty$

Consider the axiomatisations $\mathcal{E}_{\mathsf{T}}^{\downarrow}=\mathcal{E}_2\cup\{\mathrm{T},\mathrm{TP}\}$, and $\mathcal{E}_{\mathsf{S}}^{\downarrow}=\mathcal{E}_2\cup\{\mathrm{P0d},\mathrm{S},\mathrm{SP1},\mathrm{SP2}\}$, where

$$\mathrm{P0d}\qquad (x+y)\,\|\,\mathbf{0}\approx x\,\|\,\mathbf{0}+y\,\|\,\mathbf{0}$$

is a new axiom, sound modulo $\simeq_{\mathsf{B}}^{\downarrow}$. From [10] we know that, when considering BSP terms, the axiom system $\{\mathrm{A0},\mathrm{A1},\mathrm{A2},\mathrm{A3},\mathrm{T}\}$ is ground-complete modulo $\simeq_{\mathsf{T}}^{\downarrow}$ and $\{\mathrm{A0},\mathrm{A1},\mathrm{A2},\mathrm{A3},\mathrm{S}\}$ is

ground-complete modulo $\simeq_{\mathsf{S}}^{\downarrow}$. Hence, to establish the ground-completeness of $\mathcal{E}_{\mathsf{T}}^{\downarrow}$ and $\mathcal{E}_{\mathsf{S}}^{\downarrow}$, it suffices to show that all occurrences of $\|$ from $\mathrm{BSP}_{\|}$ processes can be eliminated using their respective axioms. In the remainder of this section, let $\mathsf{X} \in \{\mathsf{T}, \mathsf{S}\}$.

▶ **Lemma 40.** *For all* BSP *processes p and q there exists a closed* BSP *term r s.t. $\mathcal{E}_{\mathsf{X}}^{\downarrow} \vdash p\|q \approx r$. Then, for every* $\mathrm{BSP}_{\|}$ *process p there exists a* BSP *process q s.t. $\mathcal{E}_{\mathsf{X}}^{\downarrow} \vdash p \approx q$.*

▶ **Theorem 41.** *The axiomatisation $\mathcal{E}_{\mathsf{X}}^{\downarrow}$ is ground-complete modulo $\simeq_{\mathsf{X}}^{\downarrow}$.*

Consider the axiomatisation $\mathcal{E}_{\mathsf{X}}^{\downarrow'} = \mathcal{E}_{\mathsf{T}}^{\downarrow} \cup \{\mathrm{P2}, \mathrm{P3}\}$. We can use the technique of inverted substitutions to prove the ω-completeness of $\mathcal{E}_{\mathsf{X}}^{\downarrow'}$. To this end, it is enough to extend the mapping R defined in Section 4.3 with $R(\mathbf{1}) = \mathbf{1}$, and consider the closed substitution $\tilde{\rho}(x) = a_x.\mathbf{1}$ in the proof of the first property of the mapping.

▶ **Theorem 42.** *If $|A| = \infty$, then the axiomatisation $\mathcal{E}_{\mathsf{X}}^{\downarrow'}$ is ω-complete modulo $\simeq_{\mathsf{X}}^{\downarrow}$.*

6 Concluding remarks

We have studied the existence of finite bases over $\mathrm{BCCSP}_{\|}$ terms with respect to the behavioural equivalences in van Glabbeek's linear time-branching time spectrum. For the behavioural semantics between ready simulation and completed traces we proved a negative result for both the preorders and equivalences using Moller's proof-theoretic technique. For traces and simulation we obtained a negative result with $|A| = 1$ by applying the transformation technique to the max-plus algebra on natural numbers, for which it is known that no finite basis exists. Additionally, in the case of $2 \leq |A| < \infty$ we proved a negative result for both the preorders and equivalences using the proof-theoretic technique. Conversely, we exploited Groote's technique of inverted substitutions to obtain finite basis for trace and simulation semantics when $|A| = \infty$.

In our investigations, we independently developed the results for the preorders and their corresponding equivalences. While we encountered many similarities, it was not possible to automatically lift the results obtained for one to the other. In [6], it is shown that, under certain conditions, it is possible to lift an axiomatisation for a preorder to the corresponding equivalence over languages without parallel composition. It is an interesting avenue for future research to see whether the lifting technique can be extended to languages including a parallel composition operator and which kind of conditions will be necessary in that case.

The study of (ground-)completeness for possible futures and 2-nested simulation with $|A| = 1$ is a natural venue for future work. Additionally, for the behavioural equivalences between ready simulation and failures, in the case of $|A| = \infty$, it is unknown whether finite, sound and ground-complete axiomatisations exist. Moreover, the axiomatisability of parallel composition modulo possible worlds and impossible futures have not been studied yet [3, 14, 16]. In this paper, we considered the interleaving parallel composition operator, so the next step will be to include communication as well. In [3], it has been proved that the finite, sound and ground-complete axiomatisations over interleaving provided in [2] are preserved if CCS full merge operator is considered. It would be interesting to see whether a similar extension can be obtained for our results regarding finite bases. Furthermore, following [1], CSP-style and ACP-style communication can be studied as well.

Finally, given the amount of negative results that we obtained, it is natural to wonder whether the use of auxiliary operators can help us to find some finite bases. Given its successful application in the case of bisimilarity [9], we will first investigate the role of left merge in obtaining completeness results for all the congruences that we considered in this paper.

References

1 Luca Aceto, Elli Anastasiadi, Valentina Castiglioni, and Anna Ingólfsdóttir. Non-finite axiomatisability results via reductions: CSP parallel composition and CCS restriction. In *A Journey from Process Algebra via Timed Automata to Model Learning*, volume 13560 of *Lecture Notes in Computer Science*, pages 1–26. Springer, 2022. doi:10.1007/978-3-031-15629-8_1.

2 Luca Aceto, Valentina Castiglioni, Anna Ingólfsdóttir, Bas Luttik, and Mathias Ruggaard Pedersen. On the Axiomatisability of Parallel Composition: A Journey in the Spectrum. In *Proceedings of CONCUR 2020*, volume 171 of *LIPIcs*, pages 18:1–18:22. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.CONCUR.2020.18.

3 Luca Aceto, Valentina Castiglioni, Anna Ingólfsdóttir, Bas Luttik, and Mathias Ruggaard Pedersen. On the axiomatisability of parallel composition. *Log. Methods Comput. Sci.*, 18(1), 2022. doi:10.46298/LMCS-18(1:15)2022.

4 Luca Aceto, Zoltán Ésik, and Anna Ingólfsdóttir. On the two-variable fragment of the equational theory of the max-sum algebra of the natural numbers. Technical report, Aarhus University, 1999. doi:10.7146/brics.v6i22.20079.

5 Luca Aceto, Zoltán Ésik, and Anna Ingólfsdóttir. The max-plus algebra of the natural numbers has no finite equational basis. *Theoretical Computer Science*, 293(1):169–188, 2003. doi:10.1016/S0304-3975(02)00236-0.

6 Luca Aceto, Wan Fokkink, and Anna Ingólfsdóttir. Ready to preorder: Get your BCCSP axiomatization for free! In *Algebra and Coalgebra in Computer Science*, pages 65–79. Springer, 2007. doi:10.1007/978-3-540-73859-6_5.

7 Luca Aceto, Wan Fokkink, Rob van Glabbeek, and Anna Ingólfsdóttir. Nested semantics over finite trees are equationally hard. *Information and Computation*, 191(2):203–232, 2004. doi:10.1016/j.ic.2004.02.001.

8 Luca Aceto, Wan Fokkink, and Chris Verhoef. Structural operational semantics. In Jan A. Bergstra, Alban Ponse, and Scott A. Smolka, editors, *Handbook of Process Algebra*, pages 197–292. North-Holland / Elsevier, 2001. doi:10.1016/B978-044482830-9/50021-7.

9 Luca Aceto, Wan J. Fokkink, Anna Ingólfsdóttir, and Bas Luttik. A finite equational base for CCS with left merge and communication merge. *ACM Trans. Comput. Log.*, 10(1):6:1–6:26, 2009. doi:10.1145/1459010.1459016.

10 Josephus C.M. Baeten, Twan Basten, and Michel A. Reniers. *Process algebra: equational theories of communicating processes*, volume 50. Cambridge university press, 2010. doi:10.1017/CBO9781139195003.

11 Stefan Blom, Wan Fokkink, and Sumit Nain. On the axiomatizability of ready traces, ready simulation, and failure traces. In *Automata, Languages and Programming*, pages 109–118. Springer Berlin Heidelberg, 2003. doi:10.1007/3-540-45061-0_10.

12 Taolue Chen and Wan Fokkink. On finite alphabets and infinite bases III: Simulation. In *Proceedings of CONCUR 2006*, pages 421–434. Springer, 2006. doi:10.1007/11817949_28.

13 Taolue Chen, Wan Fokkink, Bas Luttik, and Sumit Nain. On finite alphabets and infinite bases. *Information and Computation*, 206(5):492–519, 2008. doi:10.1016/j.ic.2007.09.003.

14 Taolue Chen and Wan J. Fokkink. On the axiomatizability of impossible futures: Preorder versus equivalence. In *Proceedings of LICS 2008*, pages 156–165. IEEE Computer Society, 2008. doi:10.1109/LICS.2008.13.

15 Taolue Chen, Wan J. Fokkink, and Sumit Nain. On finite alphabets and infinite bases II: completed and ready simulation. In *Proceedings of FOSSACS 2006*, volume 3921 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2006. doi:10.1007/11690634_1.

16 Taolue Chen, Wan J. Fokkink, and Rob J. van Glabbeek. On the axiomatizability of impossible futures. *Log. Methods Comput. Sci.*, 11(3), 2015. doi:10.2168/LMCS-11(3:17)2015.

17 Robert de Simone. Higher-level synchronising devices in Meije-SCCS. *Theor. Comput. Sci.*, 37:245–267, 1985. doi:10.1016/0304-3975(85)90093-3.

18 Wan J. Fokkink and Sumit Nain. On finite alphabets and infinite bases: From ready pairs to possible worlds. In *Proceedings of FOSSACS 2004*, volume 2987 of *Lecture Notes in Computer Science*, pages 182–194. Springer, 2004. doi:10.1007/978-3-540-24727-2_14.

19 Wan J. Fokkink and Sumit Nain. A finite basis for failure semantics. In *Proceedings of ICALP 2005*, volume 3580 of *Lecture Notes in Computer Science*, pages 755–765. Springer, 2005. doi:10.1007/11523468_61.

20 Rob J. van Glabbeek. The Linear Time - Branching Time Spectrum I. In *Handbook of Process Algebra*, pages 3–99. Elsevier Science, 2001. doi:10.1016/B978-044482830-9/50019-9.

21 Jan Friso Groote. A new strategy for proving ω-completeness applied to process algebra. In *Proceedings of CONCUR 1990*, pages 314–331. Springer, 1990. doi:10.1007/BFb0039068.

22 Matthew Hennessy and Robin Milner. On observing nondeterminism and concurrency. In *Proceedings of ICALP 1980*, volume 85 of *Lecture Notes in Computer Science*, pages 299–309, 1980. doi:10.1007/3-540-10003-2_79.

23 Matthew Hennessy and Robin Milner. Algebraic laws for nondeterminism and concurrency. *J. ACM*, 32(1):137–161, 1985. doi:10.1145/2455.2460.

24 Charles A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978. doi:10.1145/359576.359585.

25 Robert M. Keller. Formal verification of parallel programs. *Commun. ACM*, 19(7):371–384, 1976. doi:10.1145/360248.360251.

26 Robin Milner. *A Calculus of Communicating Systems*. LNCS 92, Springer-Verlag, Berlin, 1980. doi:10.1007/3-540-10235-3.

27 Faron Moller. *Axioms for Concurrency*. PhD thesis, Department of Computer Science, University of Edinburgh, July 1989. Report CST-59-89. Also published as ECS-LFCS-89-84.

28 Faron Moller. The nonexistence of finite axiomatisations for CCS congruences. In *Proceedings of LICS 1990*, pages 142–153. IEEE Computer Society, 1990. doi:10.1109/LICS.1990.113741.

29 Gordon D. Plotkin. A structural approach to operational semantics. Technical report, Aarhus University, 1981.

30 Rowin Versteeg. From Bisimulation to Traces: The Impact of Parallel Composition on Finite Bases. Master's thesis, Eindhoven University of Technology, 2021. URL: https://research.tue.nl/en/studentTheses/from-bisimulation-to-traces.

Explainability is a Game for Probabilistic Bisimilarity Distances

Emily Vlasman
Department of Computer Science, Oxford University, UK

Anto Nanah Ji
Department of Electrical Engineering and Computer Science, York University, Toronto, Canada

James Worrell
Department of Computer Science, Oxford University, UK

Franck van Breugel
Department of Electrical Engineering and Computer Science, York University, Toronto, Canada

Abstract

We revisit a game from the literature that characterizes the probabilistic bisimilarity distances of a labelled Markov chain. We illustrate how an optimal policy of the game can explain these distances. Like the games that characterize bisimilarity and probabilistic bisimilarity, the game is played on pairs of states and matches transitions of those states. To obtain more convincing and interpretable explanations than those provided by generic optimal policies, we restrict to optimal policies that delay reaching observably inequivalent state pairs for as long as possible (called 1-maximal) while quickly reaching equivalent ones (called 0-minimal). We present iterative algorithms that compute 1-maximal and 0-minimal policies and prove an exponential lower bound for the number of iterations of the algorithm that computes 1-maximal policies.

2012 ACM Subject Classification Theory of computation → Program verification; Mathematics of computing → Probability and statistics

Keywords and phrases probabilistic bisimilarity distance, labelled Markov chain, game, policy, explainability

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2025.36

Funding *James Worrell*: EPSRC Fellowship EP/X033813/1.
Franck van Breugel: Natural Sciences and Engineering Research Council of Canada.

Acknowledgements The authors would like to thank the referees for their constructive feedback.

1 Introduction

Probabilistic bisimilarity, a fundamental notion introduced by Larsen and Skou [39], captures which states of a model with randomness are considered behaviourally equivalent. As shown by Katoen, Kemna, Zapreev, and Jansen [31], reducing a model by identifying states that are probabilistic bisimilar often speeds up probabilistic model checking. That is, the time it takes to reduce a model and subsequently check a property of the reduced model is often less than the time needed to check the property of the original model.

As shown by Giacalone, Jou, and Smolka [25], behavioural equivalences such as probabilistic bisimilarity are *not robust*. Even the smallest changes to the probabilities in the model may result in different states being identified as behaviourally equivalent and, hence, may lead to different reduced models. These models may even satisfy different properties.

Giacalone et al. suggested distances as a robust alternative to equivalences. Desharnais, Gupta, Jagadeesan, and Panangaden [14] proposed a quantitative generalization of probabilistic bisimilarity: *probabilistic bisimilarity distances* (or distances for short). To each pair

© Emily Vlasman, Anto Nanah Ji, James Worrell, and Franck van Breugel;
licensed under Creative Commons License CC-BY 4.0

of states, a real number in the unit interval $[0, 1]$ is assigned that captures the behavioural similarity of the states. The smaller this number, the more alike the states behave. As shown by Desharnais et al., states are probabilistic bisimilar if and only if their distance is zero.

During the last two decades, efficient algorithms have been developed to approximate and compute probabilistic bisimilarity distances (see, for example, the work of Tang [51]). Once we have computed that the distance between two states is, say $\frac{1}{8}$, it begs for an explanation. In this paper we address the question *how to explain probabilistic bisimilarity distances*.

Logic has been extensively used to explain behavioural equivalences. Logics have been designed such that for each pair of states that is not behaviourally equivalent there exists a formula of the logic such one state satisfies the formula and the other state does not. For example, for models with nondeterminism, such as labelled transition systems, bisimilarity, due to Milner [42] and Park [44], is a key behavioural equivalence. The Hennessy-Milner logic [27] provides a logical characterization of bisimilarity. For labelled Markov chains, which model systems with randomness, probabilistic bisimilarity is logically characterized by the probabilistic modal logic of [39]. Rady and Van Breugel [47] explained the distance of a pair of states by means of an infinite sequence of formulas of a logic. The major drawback of their approach is that the explanation is in general not finitely representable. In this paper, we use games to explain distances. As we will see, these can be finitely represented. Before introducing our game, we first review some related work from the literature as our approach is different from the commonly used Ehrenfeucht-Fraïssé-like games [19, 24], yet uses some ingredients of those games.

1.1 Bisimilarity

As was shown by Stirling (see, for example, [50]), bisimilarity for labelled transition systems can be characterized by means of a two-player game. In the literature we find different names for these players including Spoiler, Adversary, and Attacker for the first one and Duplicator, Prover, and Defender for the other player. Here, we use Spoiler and Duplicator. The game starts in a state pair (s, t). Spoiler tries to show that s and t are not bisimilar, whereas Duplicator tries to prove that they are.

The game is played in rounds. If a round starts in state pair (s, t), Spoiler chooses a state $u_1 \in \{s, t\}$ and an outgoing transition of u_1 with target, say, s'. Duplicator uses the other state $u_2 \in \{s, t\} \setminus \{u_1\}$ and chooses one of the outgoing transitions of u_2 with target, say, t' (preferably with the same label as s' – otherwise Duplicator loses). The next round of the game continues in the state pair (s', t'). The objective of Spoiler is to reach a state pair with different labels whereas Duplicator tries to avoid ever reaching such a state pair. As has been shown by Stirling [50, Section 2.4], states s and t are bisimilar if and only if Duplicator can avoid ever reaching a state pair with different labels when the game is started in state pair (s, t), no matter how Spoiler plays.

As pointed out by Fijalkow, Klin, and Panangaden [21], "this classical bisimulation game is elegant because it allows one to characterize a global property of behaviours (bisimilarity) in terms of a game whose rules only depend on local considerations."

1.2 Probabilistic Bisimilarity

Several characterizations of probabilistic bisimilarity for labelled Markov chains in terms of a game can be found in the literature. We will briefly review two of those next. The games are also played in rounds by two players. As in the game for bisimilarity, also in the games for probabilistic bisimilarity, the objective of Spoiler is to reach a state pair with different labels whereas Duplicator tries to avoid ever reaching such a state pair.

Let us first consider the game introduced by Desharnais, Laviolette, and Tracol in [17]. If the game is in state pair (s,t), Spoiler chooses a state $u_1 \in \{s,t\}$ and a set of states U_1. Duplicator uses the other state $u_2 \in \{s,t\} \setminus \{u_1\}$ and chooses a set of states U_2 such that the probability of transitioning to a state in U_1 from state u_1, that is, the sum of probabilities of the transitions from u_1 to a state in U_1, is less than or equal to the probability of transitioning to a state in U_2 from u_2. Such a U_2 always exists as Duplicator can pick the set of all states. Subsequently, Spoiler chooses $i \in \{1,2\}$ and a state $s' \in U_i$. Finally, Duplicator chooses a state $t' \in U_{3-i}$ (preferably with the same label as s' – otherwise Duplicator loses). The next round of the game starts in the state pair (s',t'). As in the bisimulation game, the objective of Spoiler is to reach a state pair with different labels whereas Duplicator tries to avoid ever reaching such a state pair. Desharnais, Laviolette, and Tracol [17, Theorem 5] proved that states s and t are probabilistic bisimilar if and only if Duplicator can avoid ever reaching a state pair with different labels when the game is started in state pair (s,t) no matter how Spoiler plays. Forejt, Jancar, Kiefer, and Worrell [23] generalize this game to a setting with both randomness and nondeterminism. Ford, Beohar, König, Milius, and Schröder [22] consider an even more general setting.

Fijalkow, Klin, and Panangaden [21] propose a slightly simpler game that characterizes probabilistic bisimilarity. If the game is in state-pair (s,t) then Spoiler chooses a set of states U such that the probability of transitioning from state s to a state in U is different from the probability of transitioning from state t to a state in U. If no such choice exists then Spoiler loses the game. Subsequently, Duplicator picks a state u that is in U and a state v that is not in U (preferably with the same label as u – otherwise Duplicator loses), and the game continues in the state pair (u,v). Fijalkow, Klin, and Panangaden [21, Theorem 8] showed that states s and t are probabilistic bisimilar if and only if Duplicator can avoid ever reaching a pair of states with different labels and does not lose, or Spoiler loses when the game is started in state pair (s,t) no matter how Spoiler plays.

Fijalkow, Klin, and Panangaden [21] mentioned that "the connection between metrics and bisimulation is well understood but it is possible that via the game one might gain a more quantitative understanding of the numerical significance of the metric." This paper confirms that this is the case.

1.3 Probabilistic Bisimilarity Distances

König and Mika-Michalski [36] generalize the game of Desharnais, Laviolette, and Tracol in two dimensions. Firstly, they consider distances instead of equivalences[1]. Secondly, they present a general framework based on the category of sets and functions, an endofunctor on that category, and coalgebras of that endofunctor. Below, we describe the game resulting from an instantiation of their framework so that it is applicable to distances for labelled Markov chains. The game starts in a triple (s,t,ϵ), where s and t are states and $\epsilon \in [0,1)$. Spoiler tries to show that the distance of s and t is greater than ϵ, whereas Duplicator tries to demonstrate that the distance is at most ϵ. Like all the games we discussed before, the game is played in rounds. Each round consists of the following steps. Spoiler chooses a state $u_1 \in \{s,t\}$ and a fuzzy set of states U_1. Duplicator uses the other state $u_2 \in \{s,t\} \setminus \{u_2\}$ and chooses a fuzzy set of states U_2 such that the expectation of transitioning from u_1 to U_1 minus expectation of transitioning from u_2 to U_2 is at most ϵ. Such a U_2 always exists.

[1] Desharnais et al. consider equivalence relations indexed by $\epsilon \in [0,1]$. Above, we presented their approach for $\epsilon = 0$.

Spoiler chooses $i \in \{1, 2\}$ and a state s'. Duplicator chooses a state t' with $U_i(s') \leq U_{3-i}(t')$. Duplicator can always choose a U_2 in the second step so that it can pick a t' in the fourth step with $U_i(s') \leq U_{3-i}(t')$. The next round starts in $(s', t', U_{3-i}(t') - U_i(s'))$. As shown by König and Mika-Michalski [36, Theorem 35 and 38], the distance of states s and t is at most ϵ if and only if Duplicator can avoid ever reaching a state pair with different labels when the game is started in (s, t, ϵ), no matter how Spoiler plays.

König, Mika-Michalski, and Schröder [37] present generic algorithms for computing strategies of both Spoiler and Duplicator.

Komorida, Katsumata, Hu, Klin, and Hasuo [35, Table 2] generalize the game of Fijalkow, Klin, and Panangaden to a quantitative setting using fibrations and coalgebras. Instantiating their framework to our setting amounts to the following game. The game starts in (s, t, ϵ). Spoiler chooses a fuzzy subset of states U such that the expectation of transitioning from state s to U and the expectation of transitioning from state t to U differ by more than ϵ. If such a U does not exist, Spoiler loses. Duplicator chooses states s' and t' as well as $\epsilon' \in [0, 1)$ such that $U(s')$ and $U(t')$ differ by more than ϵ'. As shown in [35, Theorem 5.11], the distance of states s and t is at most ϵ if and only if Duplicator can avoid ever reaching a state pair with different labels or Spoiler loses when the game is started in (s, t, ϵ), no matter how Spoiler plays.

1.4 Our Game

State pairs that have distance zero, which we call 0-*pairs*, are probabilistic bisimilar. We can explain their behavioural equivalence, and hence their zero distance, by the games discussed in Section 1.2. State pairs that have different labels, which we call 1-*pairs* as their distance is one, are observably different. Their different labelling explains their distance. It remains to explain the distances of the other state pairs, that is, those state pairs that have the same label but are not probabilistic bisimilar, which we call ?-*pairs*. We explain the distances of the ?-pairs by means of a game.

The foundation of the game that we study in this paper is an alternative characterization of the distances given by Chen, Van Breugel, and Worrell [10, Theorem 8]. This characterization underlies the algorithm of Bacci, Bacci, Larsen, and Mardare [1] to compute the distances. Tang [51] showed that their algorithm is an instance of Howard's *policy iteration* [28]. Howard's generic algorithm works on *Markov decision processes*. A Markov decision process is a $1\frac{1}{2}$-player game: one ordinary player and randomness which accounts for the remaining half. In our setting, the ordinary player takes on the role of Duplicator and the randomness embodies the Spoiler.

Tang [51, Section 5.3] defined the specific Markov decision processes so that Howard's policy iteration algorithm computes the distances. As shown by Tang [51, Section 6.1], one needs to decide probabilistic bisimilarity before running Howard's policy iteration algorithm. Probabilistic bisimilarity can be efficiently decided (see, for example, the algorithm of Derisavi, Hermanns, and Sanders [13]). Hence, at the time we start the game we know which states pairs are 0-pairs and 1-pairs.

Like Stirling's bisimilarity game, in our game we match transitions. In our setting we match parts of transitions. For example, consider the transitions of states 0 and 1 in Figure 1. The transition from state 1 to state 2 can be matched by part (of probability $\frac{3}{16}$) of the transition from 0 to 2. Similarly, the transition from 0 to 3 can be matched by part (of probability $\frac{1}{4}$) of the transition from 1 to 3 and the self loops of 0 and 1 can be matched. The parts that remain, part of the transition from 0 and 2 and part of the transition from 1 to 3, both have probability $\frac{1}{16}$ and can be matched as well. This matching is depicted in Figure 2.

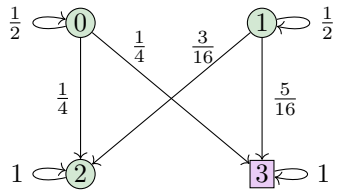


Figure 1 A labelled Markov chain.

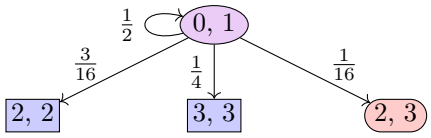


Figure 2 A coupling of the transitions of states 1 and 2 of the labelled Markov chain of Figure 1.

These matchings are also knows as *couplings*, a notion introduced by Doebling [18]. The set of couplings is known as the *transportation polytope*. There are infinitely many ways to match parts of transitions. However, as we will see, there is a finite set of couplings so that all other ones can be obtained as convex combinations of those in the finite set (the vertices of the transportation polytope). Although this set is finite, its size may be exponential in the number of states of the labelled Markov chain.

In our game the player tries match transitions, that is, chooses a coupling, in order to avoid 1-pairs. Therefore, also this game is *elegant* in that it characterizes a global property of behaviours (the distances) in terms of a game whose rules only depend on local considerations (the couplings). As we will see later, [10, Theorem 8] implies that the player can restrict to strategies, also known as *policies* in this setting, that consist of a choice of a coupling for each ?-pair. Such policies are known as deterministic and memoryless.

Given a policy, our game unfolds as follows. Let us start the game in ?-pair (s, t). The player uses their policy which provides a coupling of the transitions of s and t. Randomly, respecting the probabilities associated with the coupling, a matching of parts of transitions is chosen. This matching takes the game to a state pair (u, v). If this state pair is a 0-pair (depicted as a blue rectangle) or a 1-pair (depicted as a red rounded rectangle), the game stops. Otherwise, the game continues in ?-pair (u, v). The game reaches a 0- or 1-pair with probability one.

In the games for bisimilarity and probabilistic bisimilarity, Duplicator tried to avoid reaching 1-pairs. As our game involves randomness, the objective of the player is to minimize the probability of reaching a 1-pair. A policy is *optimal* if it minimizes the probability of reaching a 1-pair. [10, Theorem 8] shows that optimal policies exist and that for an optimal policy the probability of reaching 1-pairs from ?-pair (s, t) coincides with the distance of s and t. The coupling depicted in Figure 2 is part of an optimal policy for the labelled Markov chain of Figure 1. Note that the probability of reaching a 1-pair from ?-pair $(1, 2)$ is $\frac{1}{18}$, the distance of 1 and 2. Hence, an optimal policy can be seen as an explanation of the distances.

1.5 1-Maximal Policies

Both policies in Figure 3 are optimal and, therefore, can be seen as an explanation of the distance of states 0 and 1. In all the above described games, including ours, (one of) the player(s) tries to avoid reaching 1-pairs as only these pairs are observably different. The policy on the left matches the transitions of 0 and 1 so that none of the state pairs reachable after

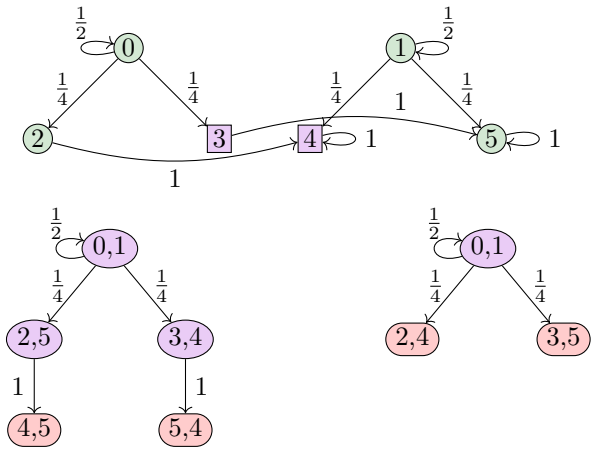


Figure 3 A labelled Markov chain and two optimal policies.

the first round of the game are observably different, whereas the one on the right matches the transitions so that two of the three reachable state pairs are 1-pairs. As a consequence, we argue that the one on the left better exemplifies the similarity of the behaviour of the states 0 and 1 and, as a result, is seen as a better explanation of the distance of 0 and 1. More generally, the larger the expected number of rounds it takes a policy to reach 1-pairs, the better it explains the distance. Therefore, policies that maximize this expected number of rounds, which we call *1-maximal*, are preferred as explanations.

By means of the algorithm of Tang [51, Section 6.2] we can compute an optimal policy. We present an iterative algorithm that, starting from an optimal policy, computes an optimal policy that is 1-maximal. We prove an exponential lower bound for the algorithm. That is, we construct a labelled Markov chain of size $O(n)$ for which the algorithm takes $\Omega(2^n)$ iterations.

1.6 0-Minimal Policies

Both policies in Figure 4 are optimal and 1-maximal. In both the ?-pair $(0, 1)$ reaches a 1-pair with probability $\frac{1}{2}$ and is expected to do so in three rounds. For 0-pairs, the transitions can be matched so that only 0-pairs are reached[2] and, hence, 1-pairs can be avoided all together. As a result, apart from avoiding 1-pairs, a secondary objective is to reach 0-pairs. The policy on the right does that better than the one of the left because its expected number of rounds to reach 0-pairs is smaller. We present an algorithm that, starting from a 1-maximal optimal policy, computes a 1-maximal optimal policy that is 0-minimal.

Let us jump into the details. Proofs and additional particulars can be found in [43, 53].

2 Probabilistic Bisimilarity Distances

We start with formalizing the model of interest, labelled Markov chains, and the probabilistic bisimilarity distances by recalling several results from the literature. Given a finite set X, a function $\mu : X \to [0, 1]$ is a *probability distribution* on X if $\sum_{x \in X} \mu(x) = 1$. We denote the set

[2] This immediately follows from the fact that probabilistic bisimilarity is a probabilistic bisimulation (see, for example, [6, Section 4]).

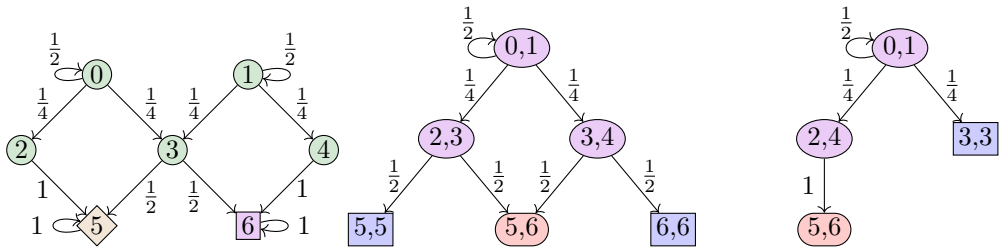


Figure 4 A labelled Markov chain and two 1-maximal optimal policies.

of probability distributions on X by $\mathcal{D}(X)$. For $\mu \in \mathcal{D}(X)$ and $A \subseteq X$, we often write $\mu(A)$ for $\sum_{x \in A} \mu(x)$. Similarly, for $\omega \in \mathcal{D}(X \times X)$, $x \in X$, and $A \subseteq X$, we usually write $\omega(x, A)$ for $\sum_{a \in A} \omega(x, a)$. For $\mu \in \mathcal{D}(X)$, we define the *support* of μ by $\mathrm{support}(\mu) = \{\, x \in X \mid \mu(x) > 0 \,\}$. To avoid clutter, for $\mu \in \mathcal{D}(X)$ and $f : X \to \mathbb{R}$ instead of $\sum_{x \in X} \mu(x)\, f(x)$ we write $\mu \cdot f$.

▶ **Definition 2.1.** *A* labelled Markov chain *is a tuple* $\langle S, L, \tau, \ell \rangle$ *consisting of*

- *a finite set* S *of* states,
- *a finite set* L *of* labels,
- *a* transition probability function $\tau : S \to \mathcal{D}(S)$, *and*
- *a* labelling function $\ell : S \to L$.

Several examples of labelled Markov chains have already been provided in the introduction. The states are represented by circles, squares, and diamonds, the labels by colours as well as shapes, and the transitions by arrows decorated with the probabilities. For the remainder of this paper, we fix a labelled Markov chain $\langle S, L, \tau, \ell \rangle$. We define probabilistic bisimilarity by means of the set $\Omega(\mu, \nu)$, which is known as the *transportation polytope* of the probability distributions μ and ν. The elements of $\Omega(\mu, \nu)$ are called *couplings*.

▶ **Definition 2.2.** *Let* $\mu, \nu \in \mathcal{D}(S)$. *The set* $\Omega(\mu, \nu)$ *is defined by*

$$\Omega(\mu, \nu) = \{\, \omega \in \mathcal{D}(S \times S) \mid \forall s \in S : \omega(s, S) = \mu(s) \wedge \omega(S, s) = \nu(s) \,\}.$$

For each $\mu, \nu \in \mathcal{D}(S)$, $\Omega(\mu, \nu)$ is a closed convex polytope. We denote the *vertices* of the transportation polytope by $V(\Omega(\mu, \nu))$.

▶ **Definition 2.3** ([30, Definition 4.3])**.** *A relation* $R \subseteq S \times S$ *is a* probabilistic bisimulation *if for all* $(s, t) \in R$, $\ell(s) = \ell(t)$ *and there exists* $\omega \in \Omega(\tau(s), \tau(t))$ *with* $\mathrm{support}(\omega) \subseteq R$. *States* s *and* t *are* probabilistic bisimilar, *denoted* $s \sim t$, *if* $(s, t) \in R$ *for some probabilistic bisimulation* R.

To define the probabilistic bisimilarity distances, it is convenient to partition the set of state pairs into 0-pairs, 1-pairs, and ?-pairs.

▶ **Definition 2.4.** *The sets* S_0^2, S_1^2, *and* $S_?^2$ *are defined by*

$$S_0^2 = \{\, (s, t) \in S \times S \mid s \sim t \,\}$$
$$S_1^2 = \{\, (s, t) \in S \times S \mid \ell(s) \neq \ell(t) \,\}$$
$$S_?^2 = (S \times S) \setminus (S_0^2 \cup S_1^2).$$

The set S_0^2 contains those state pairs that behave the same and, hence, have distance zero (see Theorem 2.6). We call these 0-pairs. The set S_1^2 contains those state pairs that have a different label and, therefore, are fundamentally differently and, hence, have distance one

(see Definition 2.5). We call these 1-pairs. The set $S^2_?$ contains the remaining state pairs. We call these ?-pairs. Note that some of these state pairs may have distance one, but cannot have distance zero (see Theorem 2.6). The probabilistic bisimilarity distances are defined in terms of the following function.

▶ **Definition 2.5.** *The function* $\Delta_1 : (S \times S \to [0,1]) \to (S \times S \to [0,1])$ *is defined by*

$$\Delta_1(d)(s,t) = \begin{cases} 0 & \text{if } (s,t) \in S^2_0 \\ 1 & \text{if } (s,t) \in S^2_1 \\ \inf\limits_{\omega \in \Omega(\tau(s),\tau(t))} \omega \cdot d & \text{otherwise.} \end{cases}$$

The functions in $S \times S \to [0,1]$ carry a natural partial order. For d, $e \in S \times S \to [0,1]$, we define $d \sqsubseteq e$ if for all s, $t \in S$, $d(s,t) \leq e(s,t)$. According to, for example, [16, Lemma 3.2], $\langle S \times S \to [0,1], \sqsubseteq \rangle$ is a complete lattice. Since the function Δ_1 is monotone (see, for example, [51, Proposition 2.1.13]), we can conclude from the Knaster-Tarski fixed point theorem [32, 52] that Δ_1 has a least fixed point, which we denote by δ_1. It maps each pair of states to a real number in the interval $[0,1]$: the *probabilistic bisimilarity distance* of the states. As we already mentioned, distance zero captures probabilistic bisimilarity.

▶ **Theorem 2.6** ([15, Theorem 4.10])**.** *For all* s, $t \in S$, $\delta_1(s,t) = 0$ *if and only if* $s \sim t$.

The 0-pairs are probabilistic bisimilar. This can be explained by means of the games mentioned in Section 1.2. The 1-pairs have different labels, which explains their difference in behaviour. Therefore, the distances of the ?-pairs remain to be explained. Hence, for the remainder of this paper, we assume that $S^2_? \neq \emptyset$ and, as a result, $S^2_1 \neq \emptyset$.

3 Policies

As we already mentioned, a policy consists of a coupling of the transitions of states s and t for each ?-pair (s,t).

▶ **Definition 3.1.** *The set* $\mathcal{P}$ *of* policies *is defined by*

$$\mathcal{P} = \{\, P \in S^2_? \to \mathcal{D}(S \times S) \mid \forall (s,t) \in S^2_? : P(s,t) \in \Omega(\tau(s), \tau(t)) \,\}.$$

Figure 3 and 4 provide examples of policies. The 0-pairs are blue rectangles, the 1-pairs are red rounded rectangles, and the ?-pairs are purple ellipses. For each ?-pair, its outgoing arrows labelled with probabilities represent a coupling. Given a policy, we can define the probability of reaching a 0- or 1-pair from any state pair as follows.

▶ **Definition 3.2.** *Let* $i \in \{0,1\}$ *and* $P \in \mathcal{P}$. *The function* $\Delta_{iP} : (S \times S \to [0,1]) \to (S \times S \to [0,1])$ *is defined by*

$$\Delta_{iP}(d)(s,t) = \begin{cases} 1 & \text{if } (s,t) \in S^2_i \\ 0 & \text{if } (s,t) \in S^2_{1-i} \\ P(s,t) \cdot d & \text{if } (s,t) \in S^2_?. \end{cases}$$

Since Δ_{iP} can be shown to be a monotone function from a complete lattice to itself, we can conclude from the Knaster-Tarski fixed point theorem that Δ_{iP} has a least fixed point. We denote the least fixed point of Δ_{iP} by δ_{iP}. For states s and t, $\delta_{iP}(s,t)$ is the probability of reaching an i-pair with respect to the policy P.

The probabilistic bisimilarity distances can be characterized in terms of a policy that minimizes the probability of reaching a 1-pair.

▶ **Theorem 3.3** ([10, Theorem 8]). $\delta_1 = \min_{P \in \mathcal{P}} \delta_{1P}$.

A policy that captures the distances, that is, the probability of reaching a 1-pair from a state pair (s,t) when using policy P equals the distance of s and t, we call optimal.

▶ **Definition 3.4.** *A policy $P \in \mathcal{P}$ is* optimal *if $\delta_{1P} = \delta_1$.*

For states s and t, the transportation polytope $\Omega(\tau(s), \tau(t))$ is generally infinite. As a result, our game, if formulated in terms of all couplings, is infinite in general. However, as we will see below, we can restrict our attention to the vertices of the transportation polytope $\Omega(\tau(s), \tau(t))$, making our game as well as the set of policies finite.

▶ **Definition 3.5.** *The set $\mathcal{V}$ of* vertex policies *is defined by*

$$\mathcal{V} = \{\, P \in \mathcal{P} \mid \forall (s,t) \in S_?^2 : P(s,t) \in V(\Omega(\tau(s), \tau(t))) \,\}.$$

Even if we restrict ourselves to vertex policies, we can still characterize the distances in terms of reachability probabilities.

▶ **Theorem 3.6** ([51, Theorem 6.1.7]). $\delta_1 = \min_{P \in \mathcal{V}} \delta_{1P}$.

According to Theorem 3.3 and 3.6, optimal and optimal vertex policies exist. We denote the set of optimal policies by $\mathcal{P}_{\text{opt}}$ and the set of optimal vertex policies by $\mathcal{V}_{\text{opt}}$. The policies depicted in Figure 3 and 4 are optimal vertex policies. The following alternative characterization of optimal policies turns out to be very useful in several of our proofs.

▶ **Proposition 3.7.** *For all $P \in \mathcal{P}$, P is optimal if and only if $\delta_1(s,t) = P(s,t) \cdot \delta_1$ for all $(s,t) \in S_?^2$.*

For every policy that is not a vertex policy, there exists a vertex policy the support graph of which is a strict subgraph. Hence, the transitions can be matched in such a way that fewer state pairs need to be considered, resulting in a simpler explanation.

▶ **Proposition 3.8.** *For all $P \in \mathcal{P}_{\text{opt}} \setminus \mathcal{V}_{\text{opt}}$ there exists $V \in \mathcal{V}_{\text{opt}}$ such that for all $(s,t) \in S_?^2$, $\text{support}(V(s,t)) \subseteq \text{support}(P(s,t))$ and $\text{support}(V(u,v)) \subset \text{support}(P(u,v))$ for some $(u,v) \in S_?^2$.*

Proof sketch. Let P be an optimal policy. For $(s,t) \in S_?^2$, we call a coupling $\omega \in \Omega(\tau(s), \tau(t))$ *optimal* for (s,t) if $\delta_1(s,t) = \omega \cdot \delta_1$. Since we can show that the set of optimal couplings for (s,t) is a closed convex polytope and the vertices of this polytope are the vertices of $\Omega(\tau(s), \tau(t))$ that are optimal, we can conclude that $P(s,t)$ is the convex combination of a set V_{st} of optimal couplings for (s,t) in $V(\Omega(\tau(s), \tau(t)))$. We can construct an optimal vertex policy V by mapping $V(s,t)$ to an optimal coupling in V_{st}. By construction, $\text{support}(V(s,t)) \subseteq \text{support}(P(s,t))$. To prove that $\text{support}(V(u,v)) \subset \text{support}(P(u,v))$ for some $(u,v) \in S_?^2$ we use the fact that a coupling is a vertex of the transportation polytope if and only if its support graph is acyclic. ◀

Tang [51, Section 6.2] presents an algorithm that computes an optimal vertex policy for a given labelled Markov chain. As we have already discussed in the introduction, although such a policy can be viewed as an explanation of the distances, we argued that 0-minimal 1-maximal optimal policies are desirable. Both 0-minimal and 1-maximal are defined in terms of expected number of rounds of the game, that is, the expected lengths of paths to 0-pairs and 1-pairs. We introduce these expected lengths next.

4 Expected Length

Let $i \in \{0,1\}$. Given an optimal policy P and a pair of states (s,t), we are interested in the expected length of paths from (s,t) to i-pairs when using P. We restrict to only those paths that reach an i-pair, that is, it is a conditional expectation. For example, consider the policy in the middle of Figure 4. To reach the 1-pair (5, 6) starting from the state pair (0, 1), there are 2 paths, each with a probability of $\frac{1}{8}$, of length 2, 2 paths, each of probability $\frac{1}{16}$, of length 3, etc. The probability of reaching a 1-pair from state pair (0, 1) is $\frac{1}{2}$. Hence, the conditional expectation is $\frac{\frac{1}{8}2+\frac{1}{8}2+\frac{1}{16}3+\frac{1}{16}3+\cdots}{\frac{1}{2}} = 3$. As we have already seen, $\delta_{1P}(s,t)$ captures the probability of all paths that reach an 1-pair from (s,t). However, when we restrict ourselves to optimal policies $\delta_{1P}(s,t) = \delta_1(s,t)$. This represents the denominator of the conditional expectation. Since P is optimal, the denominator is independent of P. Therefore, we omit it. Hence, in the following we only consider the numerator.

We define the function $\delta_0 : S \times S \to [0,1]$ by $\delta_0(s,t) = 1 - \delta_1(s,t)$. Since $\delta_{iP}(s,t)$ captures the probability of reaching an i-pair from (s,t), we can conclude that (s,t) can reach an i-pair if and only if $\delta_{iP}(s,t) \neq 0$. Because P is optimal, this is equivalent to $\delta_i(s,t) \neq 0$.

▶ **Definition 4.1.** *For $i \in \{0,1\}$, the set D_i is defined by*

$$D_i = \{\, (s,t) \in S \times S \mid \delta_i(s,t) \neq 0 \,\}.$$

Note that $S_i^2 \subseteq D_i \subseteq S_i^2 \cup S_?^2$. The expected lengths are defined in terms of the following function.

▶ **Definition 4.2.** *Let $i \in \{0,1\}$ and $P \in \mathcal{P}_{\text{opt}}$. The function $\Lambda_{iP} : (D_i \to [0,\infty)) \to (D_i \to [0,\infty))$ is defined by*

$$\Lambda_{iP}(l)(s,t) = \begin{cases} 0 & \text{if } (s,t) \in S_i^2 \\ P(s,t) \cdot (\delta_i + l) & \text{otherwise.} \end{cases}$$

Since the set S is finite and, therefore, the set D_i is finite, the set of functions $D_i \to [0,\infty)$ endowed with the sup metric[3] forms a nonempty complete metric space (see, for example, [49, Theorem 3.11]). Hence, Λ_{iP} is a function from a nonempty complete metric space to itself. To prove that Λ_{iP} has a unique fixed point, which we denote by λ_{iP}, we use the following generalization of Banach's fixed point theorem. This result dates back at least as far as the sixties. Part (a) and (b) are Banach's original fixed point theorem [4]. Part (c) can already be found in [34] and [12] contains part (d).

▶ **Theorem 4.3.** *Let (X,d) be a nonempty complete metric space. Let $f : X \to X$. If f is a power-contraction, that is, f^n is contractive for some $n \in \mathbb{N}$, then*

(a) f^n has a unique fixed point x and *(b) for all $y \in X, x = \lim_{m\in\mathbb{N}} f^{mn}(y)$.*

If f is also nonexpansive then

(c) x is the unique fixed point of f and *(d) for all $y \in X, x = \lim_{m\in\mathbb{N}} f^{m}(y)$.*

We denote P restricted to $S_?^2$ by $P_?$, that is, $P_? \in S_?^2 \to (S_?^2 \to [0,1])$.

[3] The distance of $k, l \in D_i \to [0,\infty)$ is defined as $\max_{(s,t)\in D_i} |k(s,t) - l(s,t)|$.

▶ **Proposition 4.4.** *For all $i \in \{0,1\}$, $P \in \mathcal{P}_{\mathrm{opt}}$, k, $l \in D_i \to [0,\infty)$, $(s,t) \in D_i$, and $n \in \mathbb{N}$,*
(a) *if $(s,t) \in S_i^2$ then $\Lambda_{iP}^{n+1}(k)(s,t) = \Lambda_{iP}^{n+1}(l)(s,t)$, and*
(b) *if $(s,t) \in D_i \setminus S_i^2$ then $|\Lambda_{iP}^{n+1}(k)(s,t) - \Lambda_{iP}^{n+1}(l)(s,t)| \leq P_?^n(s,t)(S_?^2)\,\|k-l\|$.*
From the above proposition we can conclude that Λ_{iP} is a power-contraction and nonexpansive.

▶ **Theorem 4.5.** *For all $i \in \{0,1\}$ and $P \in \mathcal{P}_{\mathrm{opt}}$, Λ_{iP} has unique fixed point.*

Proof sketch. Since Δ_{iP} is monotone and nonexpansive, we can conclude from [5, Theorem 1] that

$$\delta_{iP} = \lim_{n\in\mathbb{N}} \Delta_{iP}^n(\bot), \tag{1}$$

where $\bot : S \times S \to [0,1]$ is defined by $\bot(s,t) = 0$. Let $(s,t) \in D_i$. Hence, $\delta_i(s,t) \neq 0$. Since P is optimal, for $(s,t) \in D_i$ we have that $\delta_{iP}(s,t) \neq 0$. Therefore, from (1) we can conclude that exists $n_{st} \in \mathbb{N}$ such that for all $n \geq n_{st}$ we have that $\Delta_{iP}^{n+1}(\bot)(s,t) > 0$. As a consequence, we can prove that $P_?^n(s,t)(S_?^2) < 1$.

Let $n = (\max_{(s,t)\in D_i} n_{st}) + 1$ and $c = \max_{(s,t)\in D_i} P_?^n(s,t)(S_?^2)$. Note that $c < 1$. From Proposition 4.4 we can deduce that Λ_{iP}^n is c-contractive. By means of Theorem 4.3(c) we can complete the proof. ◀

The above result is a key ingredient of the proof of Proposition 5.3, which is in turn crucial in our proof of correctness and termination of Algorithm 1.

5 1-Maximal Policies

As discussed in the introduction and illustrated in Figure 3, an optimal policy that maximizes the expected length of paths to 1-pairs is desirable. This maximal expected length is defined as follows.

▶ **Definition 5.1.** *The function $\lambda_1 : D_1 \to [0,\infty]$ is defined by*

$$\lambda_1(s,t) = \sup_{P\in\mathcal{P}_{\mathrm{opt}}} \lambda_{1P}(s,t).$$

An optimal policy that realizes that maximal expected length is called 1-maximal.

▶ **Definition 5.2.** *A policy $P \in \mathcal{P}_{\mathrm{opt}}$ is* 1-maximal *if $\lambda_{1P} = \lambda_1$.*

We denote the set of optimal 1-maximal policies by $\mathcal{P}_{\mathrm{opt}}^{\max}$ and the set of optimal 1-maximal vertex policies by $\mathcal{V}_{\mathrm{opt}}^{\max}$. Similar to Proposition 3.7, we provide an alternative characterization of 1-maximal that we use in several of our proofs.

▶ **Proposition 5.3.** *For all $P \in \mathcal{P}_{\mathrm{opt}}$, P is 1-maximal if and only if $\lambda_1(s,t) = P(s,t)\cdot(\delta_1 + \lambda_1)$ for all $(s,t) \in D_1 \setminus S_1^2$.*

Below we present a policy iteration algorithm that computes a 1-maximal optimal vertex policy from an optimal vertex policy. In our algorithm we use the following function.

▶ **Definition 5.4.** *The function $\Lambda_1 : (D_1 \to [0,\infty)) \to (D_1 \to [0,\infty))$ is defined by*

$$\Lambda_1(l)(s,t) = \begin{cases} 0 & \text{if } (s,t) \in S_1^2 \\ \max\limits_{\omega\in V(\Omega(\tau(s),\tau(t)))\wedge\delta_1(s,t)=\omega\cdot\delta_1} \omega\cdot(\delta_1 + l) & \text{otherwise.} \end{cases}$$

The following theorem is crucial for proving that the vertex policy our algorithm computes is 1-maximal.

▶ **Theorem 5.5.** *λ_1 is the unique fixed point of Λ_1.*

Our algorithm starts from an optimal vertex policy P, which can be obtained by the policy iteration algorithm of [51, Section 6.2]. It computes λ_{1P} for that policy P (line 1), which can be done by means of standard algorithms (see, for example, [3, Section 10.1.1]). As long as there is a state pair in $D_1 \setminus S_1^2$ that is not locally 1-maximal with respect to the current policy (line 2), the policy at (s,t) is improved to a locally 1-maximal choice (line 3). This boils down to solving a linear programming problem. After this change to the policy P, we recompute λ_{1P} (line 4). The loop maintains the invariant that P is an optimal vertex policy as ω in line 3 is a vertex and satisfies $\delta_1(s,t) = \omega \cdot \delta_1$ (see Proposition 3.7). At termination, we have that λ_{1P} is a fixed point of Λ_1 and, by Theorem 5.5, equals λ_1 and, therefore, is 1-maximal.

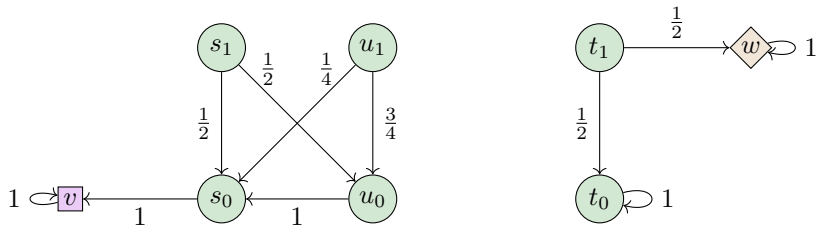
Algorithm 1 1-maximal optimal vertex policy.

Input: optimal vertex policy P
Output: 1-maximal optimal vertex policy

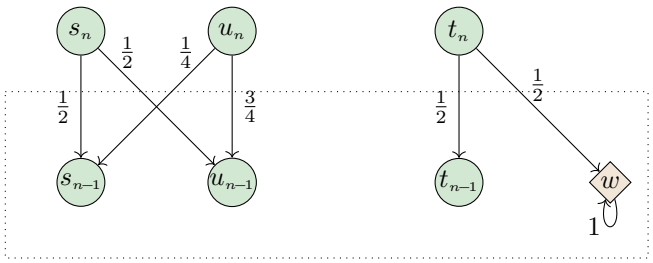
1: compute λ_{1P}
2: **while** $\exists (s,t) \in D_1 \setminus S_1^2 : \Lambda_1(\lambda_{1P})(s,t) > \lambda_{1P}(s,t)$ **do**
3: $\quad P(s,t) \leftarrow \underset{\omega \in V(\Omega(\tau(s),\tau(t))) \wedge \delta_1(s,t) = \omega \cdot \delta_1}{\arg\max} \omega \cdot (\delta_1 + \lambda_{1P})$
4: $\quad$ compute λ_{1P}
5: **end while**
6: **return** P

Next, we prove an exponential lower bound for the above algorithm.

▶ **Definition 5.6.** *The labelled Markov chain C_0 is defined as*



If $n > 0$ then the labelled Markov chain C_n is defined as



The dashed square represents the labelled Markov chain C_{n-1}.

Note that C_n has $3n+8$ states and $6n+11$ transitions and, hence, is of size $\mathcal{O}(n)$. Applying the above algorithm to the labelled Markov chain C_n results in an exponential number of iterations.

▶ **Theorem 5.7.** *For each $n \in \mathbb{N}$, the labelled Markov chain C_n of size $\mathcal{O}(n)$ is such that Algorithm 1 takes $\Omega(2^n)$ iterations.*

Proof sketch. We define a sequence of 2^n optimal vertex policies and show that there exists an execution of Algorithm 1 that cycles through all those policies. The policies only differ for the (s_i, t_i) pairs. The two transitions of s_i are matched with the two transitions of t_i in the two obvious ways. Which of those two matchings is chosen for each (s_i, t_i) pair is captured by means of the (reversed) Gray code. Hence, the exponential lower bound proof relies on the order in which the non-locally optimal (s_i, t_i) pairs are chosen (for which the current matching is replaced with the other matching) rather than the fact that the transportation polytopes may have exponentially many vertices (in our example that is not the case). ◀

6 0-Minimal Policies

As we discussed in the introduction and Figure 4 illustrates, a 1-maximal optimal policy that minimizes the expected length of paths to 0-pairs is a preferable explanation. As in the previous section, we first capture the minimal expected length.

▶ **Definition 6.1.** *The function $\lambda_0 : D_0 \to [0, \infty)$ is defined by*

$$\lambda_0(s,t) = \inf_{P \in \mathcal{P}^{\max}_{\text{opt}}} \lambda_{0P}(s,t).$$

A 1-maximal optimal policy that matches that minimal expected length is called 0-minimal.

▶ **Definition 6.2.** *A policy $P \in \mathcal{P}^{\max}_{\text{opt}}$ is* 0-minimal *if $\lambda_{0P} = \lambda_0$.*

Below we present a policy iteration algorithm that computes a 0-minimal 1-maximal optimal vertex policy from a 1-maximal optimal vertex policy. In our algorithm we use the following function.

▶ **Definition 6.3.** *The function $\Lambda_0 : (D_0 \to [0, \infty)) \to (D_0 \to [0, \infty))$ is defined by*

$$\Lambda_0(l)(s,t) = \begin{cases} 0 & \text{if } (s,t) \in S_0^2 \\ \min\limits_{\omega \in V(\Omega(\tau(s),\tau(t))) \wedge \delta_1(s,t) = \omega \cdot \delta_1 \wedge \lambda_1(s,t) = \omega \cdot (\delta_1 + \lambda_1)} \omega \cdot (\delta_0 + l) & \text{otherwise.} \end{cases}$$

The key ingredient of the correctness proof of our algorithm is the following result.

▶ **Theorem 6.4.** *λ_0 is the unique fixed point of Λ_0.*

Our algorithm that computes a 0-minimal 1-maximal vertex policy from a 1-maximal vertex policy (Algorithm 2) is very similar in structure to Algorithm 1. Instead of focussing on 1-pairs, we concentrate on 0-pairs. Furthermore, we maintain as a loop invariant that P is not only an optimal vertex policy but also that it is 1-maximal.

Algorithm 2 0-minimal 1-maximal optimal vertex policy.

Input: 1-maximal optimal vertex policy P
Output: 0-minimal 1-maximal optimal vertex policy
1: compute λ_{0P}
2: **while** $\exists (s,t) \in D_0 \setminus S_0^2 : \Lambda_0(\lambda_{0P})(s,t) < \lambda_{0P}(s,t)$ **do**
3: $\quad P(s,t) \leftarrow \underset{\omega \in V(\Omega(\tau(s),\tau(t))) \wedge \delta_1(s,t)=\omega\cdot\delta_1 \wedge \lambda_1(s,t)=\omega\cdot(\delta_1+\lambda_1)}{\arg\min} \omega \cdot (\delta_0 + \lambda_{0P})$
4: $\quad$ compute λ_{0P}
5: **end while**
6: **return** P

Algorithm 1 and 2 both optimize locally in line 3 to obtain a global optimum. This may remind the reader of the "elegance" quote of Fijalkow et al. in the introduction.

7 Symmetric Policies

Since probabilistic bisimilarity distances are symmetric, that is, $\delta_1(s,t) = \delta_1(t,s)$ for all states s and t, one may wonder whether $\delta_1(s,t)$ can be explained similarly to $\delta_1(t,s)$. We call a policy P symmetric if $P(s,t)$ and $P(t,s)$ are mirror images.

▶ **Definition 7.1.** *Let $P \in \mathcal{P}$. P is* symmetric *if for all $(s,t),(u,v) \in S_?^2$,*

$$P(s,t)(u,v) = P(t,s)(v,u).$$

We fix a total order $\prec$ on the set of states S. This allows us to turn a policy into a symmetric one as follows.

▶ **Definition 7.2.** *Let $P \in \mathcal{P}$. We define $P_\prec$ by*

$$P_\prec(s,t)(u,v) = \begin{cases} P(s,t)(u,v) & \text{if } s \prec t \\ P(t,s)(v,u) & \text{if } s \succ t. \end{cases}$$

This construction preserves all the properties of policies in which we are interested.

▶ **Theorem 7.3.** *For all $P \in \mathcal{P}$,*
(a) *$P_\prec$ is a symmetric policy,*
(b) *if P is a vertex policy then $P_\prec$ is a vertex policy,*
(c) *if P is optimal then $P_\prec$ is optimal,*
(d) *if P is 1-maximal then $P_\prec$ is 1-maximal, and*
(e) *if P is 0-minimal then $P_\prec$ is 0-minimal.*

As a result, once we have computed a 0-minimal 1-maximal optimal vertex policy P, we can turn it into a symmetric 0-minimal 1-maximal optimal vertex policy $P_\prec$. Generally, such a symmetric policy is simpler as $\delta_1(s,t)$ is explained similarly to $\delta_1(t,s)$. In the graphical representation of a symmetric policy, we only need to represent the state pairs (s,t) with $s \preceq t$, where an arrow from (s,t) to (u,v) with $u \succ v$ is depicted as a twisted arrow.

8 Experimental Results

We have implemented Algorithm 1 and 2 in Java[4]. To compute an optimal vertex policy, we used a Java implementation[5] of the algorithm presented in [51, Section 6.2]. In our experimental evaluation we used the two examples of [51, Section 9.3], namely an example of

[4] The code is available at github.com/antoNanahJi/Explainability
[5] The code is available at bitbucket.org/discoveri/probabilistic-bisimilarity-distances.

Table 1 Average (and standard deviation) in milliseconds of time to compute optimal, 1-maximal, and 0-minimal policies.

	optimal	1-maximal	0-minimal
Crowds	913 (28)	1002 (24)	1099 (27)
Leader	494 (14)	593 (19)	576 (21)
Miller-Rabin	32,149 (149)	479 (2)	1,069 (5)
Quicksort	218,232 (8,971)	38,528 (1,139)	46,416 (1,165)
Two dies	3,142 (1)	2,271 (1)	1,569 (9)

two dies due to Knuth and Yao [33] and randomized quick sort, as well as the Miller-Rabin primality test [41, 46], the crowds protocol by Reiter and Rubin in [48], and the leader election protocol by Itai and Rodeh [29].

The experiments were run on an Intel machine with an i7-8700T CPU and 15 GB of RAM. For each of the five examples, we first computed an optimal vertex policy, then a 1-maximal optimal vertex policy, and finally a 0-minimal 1-maximal optimal vertex policy. We timed 55 trails for each of the three stages of the computation and report the average (and standard deviation) in milliseconds in Table 1. Since a Java virtual machine needs to perform just-in-time compilation and optimization, we discarded the first eight trails. Garbage collection was triggered in between trials to minimize its impact on our measurements.

As can be seen, the algorithms tend to perform well in practice despite the exponential lower bound. In most cases, computing the 1-maximal and 0-minimal optimal policies is significantly faster than computing optimal policies. Recall that if policy P is optimal then λ_{1P} captures the distances. Hence, once we have computed the distances by means of policy iteration, we can construct a symmetric 0-minimal 1-maximal optimal vertex policy that explains the distances, often taking less time than computing the distances.

Our implementation also provides a graphical representation (DOT format) of the policy.

9 Conclusion

We have shown that explainability is a game in the context of probabilistic bisimilarity distances of labelled Markov chains. More precisely, symmetric 0-minimal 1-maximal optimal vertex policies for the $1\frac{1}{2}$-player games presented in this paper explain the distances.

Apart from the area of probabilistic model checking, distances similar to the ones studied in this paper also play a role in other fields including control theory [26], fault-tolerance [9], privacy [11], quantum computing [20], reinforcement learning [40], and systems-biology [38]. As a consequence, we anticipate that our results are widely applicable.

The main contributions of this paper are the following.

- We provide a unified overview of games related to probabilistic bisimilarity and probabilistic bisimilarity distances in the introduction of the paper.
- We argue that symmetric 0-minimal 1-maximal optimal vertex policies explain these distances.
- We develop algorithms to compute symmetric 0-minimal 1-maximal optimal vertex policies and prove them correct.
- We prove an exponential lower bound for the algorithm that computes a 1-maximal optimal vertex policy from an optimal vertex policy.
- We have implemented all algorithms in Java. The code is open-source.

Several questions remain open. For example, we conjecture that the algorithm that computes a 0-minimal 1-maximal optimal policy from a 1-maximal optimal policy also has an exponential lower bound. Determining upper bounds for both our algorithms is left for future work.

In [53], Vlasman introduced two properties of policies, label conflict and probabilistic bisimilar conflict, argued that policies free of such conflicts are desirable for explaining probabilistic bisimilarity distances, and presented algorithms to remove such conflicts from optimal policies. We have already shown that 1-maximal optimal policies are free of label conflicts (see [43]). It is an open problem whether a 0-minimal 1-maximal optimal policy is free of probabilistic bisimilar conflicts. We conjecture that "transitivity" of policies, which seems closely related to the fact that probabilistic bisimilarity distances satisfy the triangle inequality, may play a role in tackling that problem. Apart from symmetry and "transitivity," it could be worthwhile exploring other forms of structure of policies to further reduce the complexity of the explanation.

We conjecture that the policy iteration algorithm of [51, Section 6.2], as well as Algorithm 1 and 2 can be modified so that they maintain that the policy is symmetric as a loop invariant. This may allows us to restrict our attention to only half of the state pairs currently being considered in the algorithm.

As we mentioned in the introduction, our game can be viewed as a Markov decision process presented by Tang in [51, Section 5.3]. Several definitions and results can be expressed in the parlance of Markov decision processes. For example, the function Δ_1 presented in Definition 2.5 can be seen as a Bellman equation (see, for example, [45, Section 4.3]) – note that a slight generalization of Δ_1 was already defined in, for example, [7, Definition 15], which predates Tang's presentation of the Markov decision process by more than a decade. Similarly, for a policy P and a state pair (s,t), $P(s,t)$ can be seen a decision rule and Proposition 3.7 then characterizes a conserving decision rule (see, for example, [45, Section 6.2.4]).

Our results are related to the work of Busatto-Gaston et al. [8]. They consider bi-objective problems on Markov decision processes. In particular, they minimize the (conditional) expected number of steps to a target while guaranteeing the maximal probability of reaching it. In this paper, we consider three objectives (optimal, 1-maximal, and 0-minimal). They propose a simple two-step pruning algorithm. Our algorithms show similarities to their algorithm. For example, Definition 5.4 corresponds to pruning the Markov decision process by restricting to couplings ω that satisfy $\delta_1(s,t) = \omega \cdot \delta_1$. Since the size of the Markov decision process representing our game may be exponential in the number of states of the labelled Markov chain, we do not explicitly construct the Markov decision process, nor do we explicitly prune it.

The aim of our work is to provide a human-interpretable explanation of probabilistic bisimilarity distances. According to Theorem 3.3, for a policy P we have that δ_{1P} provides an upper bound for δ_1. As a result, a policy can also be viewed as a certificate (see, for example, [2] for work on certificates for Markov decision processes) for an upper bound of the probabilistic bisimilarity distances. In this case, we would aim for a small policy P so that δ_{1P} can be computed quickly. We consider this an interesting line of future work.

References

1 Giorgio Bacci, Giovanni Bacci, Kim Larsen, and Radu Mardare. On-the-fly exact computation of bisimilarity distances. In Nir Piterman and Scott Smolka, editors, *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of*

Systems, volume 7795 of *Lecture Notes in Computer Science*, pages 1–15, Rome, Italy, March 2013. Springer-Verlag. doi:10.1007/978-3-642-36742-7_1.

2 Christel Baier, Calvin Chau, and Sascha Klüppelholz. Certificates and witnesses for multi-objective queries in Markov decision processes. In Jane Hillston, Sadegh Soudjani, and Masaki Waga, editors, *Proceedings of the 1st International Joint Conference on Quantitative Evaluation of Systems and Formal Modeling and Analysis of Timed Systems*, volume 14996 of *Lecture Notes in Computer Science*, pages 1–18, Calgary, Canada, September 2024. Springer-Verlag. doi:10.1007/978-3-031-68416-6_1.

3 Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, Cambridge, MA, USA, 2008.

4 Stefan Banach. Sur les opérations dans les ensembles abstraits et leurs applications aux equations intégrales. *Fundamenta Mathematicae*, 3:133–181, 1922. doi:10.4064/fm-3-1-133-181.

5 Franck van Breugel. On behavioural pseudometrics and closure ordinals. *Information Processing Letters*, 112(19):715–718, October 2012. doi:10.1016/J.IPL.2012.06.019.

6 Franck van Breugel. Probabilistic bisimilarity distances. *SIGLOG News*, 4(4):33–51, October 2017. doi:10.1145/3157831.3157837.

7 Franck van Breugel, Claudio Hermida, Michael Makkai, and James Worrell. An accessible approach to behavioural pseudometrics. In Luís Caires, Giuseppe Italiano, Luís Monteiro, Catuscia Palamidessi, and Moti Yung, editors, *Proceedings of the 32nd International Colloquium on Automata, Languages and Programming*, volume 3580 of *Lecture Notes in Computer Science*, pages 1018–1030, Lisbon, Portugal, July 2005. Springer-Verlag. doi:10.1007/11523468_82.

8 Damien Busatto-Gaston, Debraj Chakraborty, Anirban Majumdar, Sayan Mukherjee, Guillermo Pérez, and Jean-François Raskin. Bi-objective lexicographic optimization in Markov decision processes with related objectives. In Étienne André and Jun Sun, editors, *Proceedings of the 21st International Symposium on Automated Technology for Verification and Analysis*, volume 14215 of *Lecture Notes in Computer Science*, pages 203–223, Signapore, October 2023. Springer-Verlag. doi:10.1007/978-3-031-45329-8_10.

9 Pablo Castro, Pedro D'Argenio, Ramiro Demasi, and Luciano Putruele. Measuring masking fault-tolerance. In *Proceedings of the 25th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 11428 of *Lecture Notes in Computer Science*, pages 375–392, Prague, Czech Republic, April 2019. Springer-Verlag. doi:10.1007/978-3-030-17465-1_21.

10 Di Chen, Franck van Breugel, and James Worrell. On the complexity of computing probabilistic bisimilarity. In Lars Birkedal, editor, *Proceedings of the 15th International Conference on Foundations of Software Science and Computational Structures*, volume 7213 of *Lecture Notes in Computer Science*, pages 437–451, Tallinn, Estonia, March/April 2012. Springer-Verlag. doi:10.1007/978-3-642-28729-9_29.

11 Dmitry Chistikov, Andrzej Murawski, and David Purser. Bisimilarity distances for approximate differential privacy. In Shuvendu Lahiri and Chao Wang, editors, *Proceedings of the 16th International Symposium on Automated Technology for Verification and Analysis*, volume 11138 of *Lecture Notes in Computer Science*, pages 194–210, Los Angeles, CA, USA, October 2018. Springer-Verlag. doi:10.1007/978-3-030-01090-4_12.

12 Eric Denardo. Contraction mappings in the theory underlying dynamic programming. *SIAM Review*, 9(2):165–177, April 1967. doi:10.1137/1009030.

13 Salem Derisavi, Holger Hermanns, and William Sanders. Optimal state-space lumping in Markov chains. *Information Processing Letters*, 87(6):309–315, September 2003. doi:10.1016/S0020-0190(03)00343-0.

14 Josée Desharnais, Vineet Gupta, Radha Jagadeesan, and Prakash Panangaden. Metrics for labeled Markov systems. In Jos Baeten and Sjouke Mauw, editors, *Proceedings of 10th International Conference on Concurrency Theory*, volume 1664 of *Lecture Notes in Computer Science*, pages 258–273, Eindhoven, the Netherlands, August 1999. Springer-Verlag. doi:10.1007/3-540-48320-9_19.

15 Josée Desharnais, Vineet Gupta, Radha Jagadeesan, and Prakash Panangaden. Metrics for labelled Markov processes. *Theoretical Computer Science*, 318(3):323–354, June 2004. doi:10.1016/J.TCS.2003.09.013.

16 Josée Desharnais, Radha Jagadeesan, Vineet Gupta, and Prakash Panangaden. The metric analogue of weak bisimulation for probabilistic processes. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pages 413–422, Copenhagen, Denmark, July 2002. IEEE. doi:10.1109/LICS.2002.1029849.

17 Josée Desharnais, François Laviolette, and Mathieu Tracol. Approximate analysis of probabilistic processes: logic, simulation and games. In *Proceedings of the 5th International Conference on the Quantitative Evaluation of Systems*, pages 264–273, Saint-Malo, France, September 2008. IEEE. doi:10.1109/QEST.2008.42.

18 Wolfgang Doeblin. Exposé de la théorie des chaînes simples constantes de Markov à un nombre fini d'états. *Revue mathématique de l'union interbalkanique*, 2:77–105, 1938.

19 Andrzej Ehrenfeucht. An application of games to the completeness problem for formalized theories. *Fundamenta Mathematicae*, 49:129–141, 1961.

20 Yuan Feng, Runyao Duan, and Mingsheng Ying. Bisimulation for quantum processes. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 523–534, Austin, TX, USA, January 2011. ACM. doi:10.1145/1926385.1926446.

21 Nathanaël Fijalkow, Bartek Klin, and Prakash Panangaden. Expressiveness of probabilistic modal logics, revisited. In Ioannis Chatzigiannakis, Piotr Indyk, Fabian Kuhn, and Anca Muscholl, editors, *Proceedings of 44th International Colloquium on Automata, Languages and Programming*, volume 80 of *Leibniz International Proceedings in Informatics*, pages 105:1–105:12, Warsaw, Poland, July 2017. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPICS.ICALP.2017.105.

22 Chase Ford, Harsh Beohar, Barbara König, Stefan Milius, and Lutz Schröder. Graded monads and behavioural equivalence games. In Christel Baier and Dana Fisman, editors, *Proceedings of the 37th Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 61:1–61:13, Haifa, Israel, August 2022. ACM. doi:10.1145/3531130.3533374.

23 Vojtech Forejt, Petr Jancar, Stefan Kiefer, and James Worrell. Game characterization of probabilistic bisimilarity and applications to pushown automata. *Logical Methods in Computer Science*, 14(4:13), November 2018. doi:10.23638/LMCS-14(4:13)2018.

24 Roland Fraïssé. Sur une nouvelle classification des systèmes de relations. *Comptes Rendus*, 230:1022–1024, 1950.

25 Alessandro Giacalone, Chi-Chang Jou, and Scott Smolka. Algebraic reasoning for probabilistic concurrent systems. In *Proceedings of the IFIP WG 2.2/2.3 Working Conference on Programming Concepts and Methods*, pages 443–458, Sea of Gallilee, April 1990. North-Holland.

26 Antoine Girard and George Pappas. Approximate bisimulation: A bridge between computer science and control theory. *European Journal of Control*, 17(5):568–578, September 2011. doi:10.3166/EJC.17.568-578.

27 Matthew Hennessy and Robin Milner. On observing nondeterminism and concurrency. In Jaco de Bakker and Jan van Leeuwen, editors, *Proceedings of 7th International Colloquium on Automata, Languages and Programming*, volume 85 of *Lecture Notes in Computer Science*, pages 299–309, Noordwijkerhout, the Netherlands, July 1980. Springer-Verlag. doi:10.1007/3-540-10003-2_79.

28 Ronald Howard. *Dynamic programming and Markov processes*. The MIT Press, Cambridge, MA, USA, 1960.

29 Alon Itai and Michael Rodeh. Symmetry breaking in distributed networks. In *Proceedings of the 2nd Annual Symposium on Foundations of Computer Science*, pages 150–158, Nashville, TN, USA, October 1981. IEEE. doi:10.1109/SFCS.1981.41.

30 Bengt Jonsson and Kim Larsen. Specification and refinement of probabilistic processes. In *Proceedings of the 6th Annual Symposium on Logic in Computer Science*, pages 266–277, Amsterdam, the Netherlands, July 1991. IEEE. doi:10.1109/LICS.1991.151651.

31 Joost-Pieter Katoen, Tim Kemna, Ivan Zapreev, and David Jansen. Bisimulation minimisation mostly speeds up probabilistic model checking. In Orna Grumberg and Michael Huth, editors, *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 4424 of *Lecture Notes in Computer Science*, pages 87–101, Braga, Portugal, March/April 2007. Springer-Verlag. doi:10.1007/978-3-540-71209-1_9.

32 Bronisław Knaster. Un théorème sur les fonctions d'ensembles. *Annales Polonici Mathematici*, 6:133–134, 1928.

33 Donald Knuth and Andrew Yao. The complexity of nonuniform random number generation. In Joseph Traub, editor, *Proceedings of a Symposium on New Directions and Recent Results in Algorithms and Complexity*, pages 375–428, Pittsburgh, PA, USA, April 1976. Academic Press.

34 Ignace Kolodner. Fixed points. *The American Mathematical Monthly*, 71(9):906, October 1964.

35 Yuichi Komorida, Shin-ya Katsumata, Nick Hu, Bartek Klin, and Ichiro Hasuo. Codensity games for bisimilarity. In *Proceedings of the 34th Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 1–13, Vancouver, Canada, June 2019. IEEE. doi:10.1109/LICS.2019.8785691.

36 Barbara König and Christina Mika-Michalski. (Metric) bisimulation games and real-valued modal logics for coalgebras. In Sven Schewe and Lijun Zhang, editors, *Proceedings of the 29th International Conference on Concurrency Theory*, volume 118 of *Leibniz International Proceedings in Informatics*, pages 37:1–37:17, Beijing, China, September 2018. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPICS.CONCUR.2018.37.

37 Barbara König, Christina Mika-Michalski, and Lutz Schröder. Explaining non-bisimilarity in a coalgebraic approach: Games and distinguishing formulas. In Daniela Petrisan and Jurriaan Rot, editors, *Proceedings of the 5th IFIP WG 1.3 International Workshop on Coalgebraic Methods in Computer Science*, volume 12094 of *Lecture Notes in Computer Science*, pages 133–154, Dublin, Ireland, April 2020. Springer-Verlag. doi:10.1007/978-3-030-57201-3_8.

38 Ilya Korsunsky, Kathleen McGovern, Tom LaGatta, Loes Olde Loohuis, Terri Grosso-Applewhite, Nancy Griffeth, and Bud Mishra. Systems biology of cancer: a challenging expedition for clinical and quantitative biologists. *Frontiers in Bioengineering and Biotechnology*, 2, August 2014. doi:10.3389/fbioe.2014.00027.

39 Kim Larsen and Arne Skou. Bisimulation through probabilistic testing. In *Proceedings of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 344–352, Austin, TX, USA, January 1989. ACM. doi:10.1145/75277.75307.

40 Qiyuan Liu, Qi Zhou, Rui Yang, and Jie Wang. Robust representation learning by clustering with bisimulation metrics for visual reinforcement learning with distractions. In Brian Williams, Yiling Chen, and Jennifer Neville, editors, *Proceedings of the AAAI Conference on Artificial Intelligence*, Washington, CD, USA, February 2023. doi:10.1609/aaai.v37i7.26063.

41 Gary Miller. Riemann's hypothesis and tests for primality. In William Rounds, Nancy Martin, Jack Carlyle, and Michael Harrison, editors, *Proceedings of the 7th Annual ACM Symposium on Theory of Computing*, pages 234–239, Albuquerque, NM, USA, May 1975. ACM. doi:10.1145/800116.803773.

42 Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Germany, 1980. doi:10.1007/3-540-10235-3.

43 Anto Nanah Ji. Explainabilty is a game for probabilistic bisimilarity distances. Master's thesis, York University, Toronto, Canada, May 2025.

44 David Park. Concurrency and automata on infinite sequences. In Peter Deussen, editor, *Proceedings of 5th GI-Conference on Theoretical Computer Science*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183, Karlsruhe, Germany, March 1981. Springer-Verlag. doi:10.1007/BFB0017309.

45 Martin Puterman. *Markov Decision Processes*. Wiley, Hoboken, NJ, USA, 1994.

46 Michael Rabin. Probabilistic algorithm for testing primality. *Journal of Number Theory*, 12(1):128–138, February 1980. doi:10.1016/0022-314X(80)90084-0.

47 Amgad Rady and Franck van Breugel. Explainability of probabilistic bisimilarity distances for labelled Markov chains. In Orna Kupferman and Pawel Sobocinski, editors, *Proceedings of the 26th International Conference on Foundations of Software Science and Computational Structures*, volume 13992 of *Lecture Notes in Computer Science*, pages 285–307, Paris, France, April 2023. Springer-Verlag. doi:10.1007/978-3-031-30829-1_14.

48 Michael Reiter and Aviel Rubin. Crowds: anonymity for web transactions. *ACM Transactions on Information and System Security*, 1(1):66–92, November 1998. doi:10.1145/290163.290168.

49 Walter Rudin. *Principles of Mathematical Analysis*. McGraw Hill, New York, NY, USA, 1976.

50 Colin Stirling. Modal and temporal logics for processes. In Faron Moller and Graham Birtwistle:, editors, *Proceedings of 8th Banff Higher Order Workshop*, volume 1043 of *Lecture Notes in Computer Science*, pages 149–237, Banff, Canada, August/September 1995. Springer-Verlag. doi:10.1007/3-540-60915-6_5.

51 Qiyi Tang. *Computing probabilistic bisimilarity distances*. PhD thesis, York University, Toronto, Canada, August 2018.

52 Alfred Tarski. A lattice-theoretic fixed point theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, June 1955. doi:10.2140/pjm.1955.5.285.

53 Emily Vlasman. On logical and game characterizations of probabilistic bisimilarity distances for labelled Markov chains. Master's thesis, University of Oxford, Oxford, UK, September 2023.

Prophecies All the Way: Game-Based Model-Checking for HyperQPTL Beyond ∀*∃*

Sarah Winter
IRIF, Université Paris Cité, Paris, France

Martin Zimmermann
Aalborg University, Denmark

Abstract

Model-checking HyperLTL, a temporal logic expressing properties of sets of traces with applications to information-flow based security and privacy, has a decidable, but TOWER-complete, model-checking problem. While the classical model-checking algorithm for full HyperLTL is automata-theoretic, more recently, a game-based alternative for the $\forall^*\exists^*$-fragment has been presented.

Here, we employ imperfect information-games to extend the game-based approach to full HyperQPTL, which features arbitrary quantifier prefixes and quantification over propositions and can express every ω-regular hyperproperty. As a byproduct of our game-based algorithm, we obtain finite-state implementations of Skolem functions via transducers with lookahead that explain satisfaction or violation of HyperQPTL properties.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Modal and temporal logics; Theory of computation → Verification by model checking

Keywords and phrases HyperLTL, HyperQPTL, model-checking games, prophecies

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2025.37

Related Version *Full Version*: `https://arxiv.org/abs/2504.08575` [25]

Funding *Martin Zimmermann*: Supported by DIREC - Digital Research Centre Denmark.

1 Introduction

Hyperlogics like HyperLTL and HyperCTL* [6] extend their classical counterparts LTL [20] and CTL* [10] by trace quantification and are thereby able to express so-called hyperproperties [7], properties which relate multiple execution traces of a system. These are crucial for expressing information-flow properties capturing security and privacy requirements [6]. For example, generalized noninterference [19] is captured by the HyperLTL formula

$$\varphi_{\text{GNI}} = \forall\pi.\ \forall\pi'.\ \exists\pi''.\ \mathbf{G}\left(\bigwedge\nolimits_{\mathtt{p}\in L_{\text{in}}\cup L_{\text{out}}} \mathtt{p}_\pi \leftrightarrow \mathtt{p}_{\pi''}\right) \wedge \mathbf{G}\left(\bigwedge\nolimits_{\mathtt{p}\in H_{\text{in}}} \mathtt{p}_{\pi'} \leftrightarrow \mathtt{p}_{\pi''}\right)$$

expressing that for all traces π and π' there exists a trace π'' that agrees with the low-security inputs (propositions in L_{in}) and low-security outputs (propositions in L_{out}) of π and the high-security inputs (propositions in H_{in}) of π'. Intuitively, it is satisfied if every input-output behavior observable by a low-security user of a system is compatible with any sequence of high-security inputs, i.e., the low security input-output behavior does not leak information about the high-security inputs, which should indeed be unobservable (directly and indirectly) by a low-security user.

These expressive logics offer a uniform approach to the specification, analysis, and verification of hyperproperties with intuitive syntax and semantics, and a decidable [6], albeit TOWER-complete [22, 18], model-checking problem. The classical model-checking algorithm [6] is automata-based and even the case of $\forall^*\exists^*$-formulas like φ_{GNI} requires the complementation of ω-automata, a famously challenging construction to implement.

© Sarah Winter and Martin Zimmermann;
licensed under Creative Commons License CC-BY 4.0

As an alternative, it is beneficial to draw upon the deep connections between logic and games. Consider a HyperLTL formula of the form $\forall\pi.\ \exists\pi'.\psi$ with quantifier-free ψ. It is straightforward to capture the semantics of HyperLTL by a model-checking game between two players called Verifier (trying to prove $\mathfrak{T} \models \varphi$) and Falsifier (trying to prove $\mathfrak{T} \not\models \varphi$). In the model-checking game, Falsifier picks a trace of $\mathfrak{T}$ to interpret π and then Verifier picks a trace of $\mathfrak{T}$ to interpret π'. Verifier wins if these two traces satisfy the formula ψ, otherwise Falsifier wins. The game can easily be shown sound (if Verifier wins, then $\mathfrak{T} \models \varphi$) and complete (if $\mathfrak{T} \models \varphi$ then Verifier wins), as winning strategies for Verifier are Skolem functions for π' and vice versa. But the game has one major drawback: both players have in general uncountably many possible moves as they are picking traces.

To obtain a game that can be handled algorithmically, Coenen et al. introduced the following variation of the model-checking game [8] (which we will call the alternating game): Instead of picking the traces of $\mathfrak{T}$ in one move, the players construct them alternatingly one vertex at a time. Together with a deterministic ω-automaton that is equivalent to ψ, this yields a finite game with ω-regular winning condition that is sound, but possibly incomplete ($\mathfrak{T} \models \varphi$ does not always imply that Verifier wins the alternating game).

The problem boils down to the informedness of Verifier: In the model-checking game, she has knowledge about the full trace assigned to π when picking π'. On the other hand, in the alternating game, Verifier only has access to the first vertex of the trace assigned to π when picking the first vertex of the trace assigned to π', which puts her at a disadvantage. That the alternating game can be incomplete is witnessed by the formula

$$\varphi_{\text{inc}} = \forall\pi.\ \exists\pi'.\ \mathtt{p}_{\pi'} \leftrightarrow \mathbf{F}\,\mathtt{p}_{\pi}$$

expressing that for every trace π in $\mathfrak{T}$ there is a trace π' in $\mathfrak{T}$ such that $\mathtt{p}$ holds at the first position of π' if and only if there is a $\mathtt{p}$ somewhere in π, i.e., the first letter of π' depends (possibly) on every letter of π. So, by not picking a $\mathtt{p}$ in the first round, Falsifier forces Verifier to make a prediction about whether Falsifier will ever pick a $\mathtt{p}$ or not in the future. However, Falsifier can easily contradict that prediction and thereby win, e.g., for a transition system $\mathfrak{T}_{\text{all}}$ having all traces over $\mathtt{p}$. Thus, we indeed have $\mathfrak{T}_{\text{all}} \models \varphi$, but Verifier does not win the alternating game.

However, a single bit of lookahead allows Verifier to win the alternating game induced by $\mathfrak{T}_{\text{all}}$ and φ_{inc}, i.e., the answer to the query "will the trace picked by Falsifier contain a $\mathtt{p}$". If Falsifier has to provide this information with his first move, then Verifier can make her first move accordingly. For correctness, we additionally have to adapt the rules of the alternating game so that Falsifier loses when contradicting his answer made during the first round, e.g., if he commits to there being no $\mathtt{p}$'s in his trace, but then picking one.

In general, it is not sufficient to ask a single query at the beginning of a play, but one needs to ask queries at every move of Falsifier. To see this, let us consider another example, this time with the formula

$$\varphi'_{\text{inc}} = \forall\pi.\ \exists\pi'.\ \mathbf{G}(\mathtt{p}_{\pi'} \leftrightarrow \mathbf{X}\,\mathtt{p}_{\pi}).$$

Here, Verifier has to always pick a $\mathtt{p}$ in her move if and only if Falsifier will pick a $\mathtt{p}$ in his next move (which Verifier does not have yet access to). Thus, Verifier wins if she gets the (truthful) answer to the query "does the next move by Falsifier contain a $\mathtt{p}$", but loses without the ability to query Falsifier. Note that in both cases, the queries are used to obtain a (binding) commitment about the future moves that Falsifier will make.

Coenen et al. [8] and Beutner and Finkbeiner [5] showed how to formalize this intuition using so-called prophecies [1]. A single prophecy is a language P of traces and is associated with a (Boolean) prophecy variable. Now, in addition to picking, vertex by vertex, a trace of

$\mathfrak{T}$, Falsifier also picks a truth value for the prophecy variable with the interpretation that a value of 1 corresponds to the suffix of the trace picked by him from now on being in P and that a value of of 0 corresponds to the suffix of the trace picked by him from now on not being in P. If the language P is ω-regular, then one can employ an ω-automaton to check whether the predictions made by Falsifier are actually truthful (and make him lose if they are not truthful). We call the resulting game the alternating game with prophecies. Intuitively, it constitutes a middle-ground between the naive game where both players pick traces and the alternating game (without prophecies). The former is sound and complete, but not finite-state while the latter is sound and finite-state, but not complete.

The alternating game with (finitely many) ω-regular prophecies is sound and finite-state and Beutner and Finkbeiner showed that for every $\forall^*\exists^*$-formula there is a finite and effectively computable set of ω-regular prophecies such that Verifier wins the alternating game with these prophecies if and only if $\mathfrak{T} \models \varphi$, i.e., there are always prophecies that make the alternating game-based approach to model-checking also complete. Furthermore, the resulting game with prophecies has a ω-regular winning condition, and can therefore be solved effectively. Note that this construction is still automata-based, as the prophecies are derived from a (deterministic) ω-automaton for the quantifier-free part of φ. Beutner and Finkbeiner implemented their prophecy-based model-checking algorithm and presented encouraging results on small instances, e.g., their prototype implementation is in some cases the first tool that can prove $\mathfrak{T} \not\models \varphi$, a result that was out of reach for existing model-checking tools [5]. These results shows the potential of the game-based approach to HyperLTL model-checking.

However, one question remains open: can the alternating game-based approach be extended to full HyperLTL, i.e., beyond a single quantifier alternation. There is precedent for such a game-based analysis of full HyperLTL: Recently, Winter and Zimmermann showed that the existence of (Turing) computable Skolem functions for existentially quantified variables can be characterized by a game [24]. For example, $\mathfrak{T}_{\text{all}} \models \varphi'_{\text{inc}}$ is witnessed by computable Skolem functions, as reading a prefix of length n of π allows to compute the prefix of length $n-1$ of π' so that for every π, π and the resulting π' satisfy $\mathbf{G}(\mathtt{p}_{\pi'} \leftrightarrow \mathbf{X}\,\mathtt{p}_{\pi})$. On the other hand, $\mathfrak{T}_{\text{all}} \models \varphi_{\text{inc}}$ is not witnessed by computable Skolem functions, as the choice of the first letter of π' depends, as explained above, on all letters of π'. Such a Skolem function is not computable by a Turing machine, as it is not continuous.

The game characterizing the existence of computable Skolem functions, say for a formula

$$\varphi = \forall \pi_0.\ \exists \pi_1.\ \ldots \forall \pi_{k-2}.\ \exists \pi_{k-1}.\ \psi$$

with quantifier-free ψ, is a multi-player game played between a player in charge of selecting, vertex by vertex, a trace for each universally quantified variable (i.e., he has the role that Falsifier has in the previous games), and a coalition of players, one for each existentially quantified variable (i.e., the coalition has the role that Verifier has in the previous games). Furthermore, the game must be of imperfect information in order to capture the semantics of HyperLTL, where the choice of π_i may only depend on the choice of the π_j with $j < i$. Thus, in the game, the player in charge of an existentially quantified π_i only has access to the choices made so far for the π_j for $j < i$. Furthermore, the game needs to incorporate a delay [16, 17, 12] between the moves of the different players to capture the fact that a choice by one of the existential players may depend on future moves by the universal player (see, e.g., the formula φ'_{inc} above). The main insight then is that a bounded delay is always sufficient, if there are computable Skolem functions at all (see, again, the difference between φ_{inc} (which has no computable Skolem functions over $\mathfrak{T}_{\text{all}}$) and φ'_{inc} (which has computable Skolem functions over $\mathfrak{T}_{\text{all}}$)).

Our Contribution. We present the first effective game-based characterization of model-checking for full HyperLTL (and even HyperQPTL, which allows to express all ω-regular hyperproperties [22, 13]), yielding a sound and complete imperfect information finite-state game with ω-regular winning condition. This result generalizes both the alternating game with prophecies from $\forall^*\exists^*$-formulas to formulas with arbitrary quantifier prefixes and the game of Winter and Zimmermann from characterizing the existence of computable Skolem functions witnessing $\mathfrak{T} \models \varphi$ to characterizing $\mathfrak{T} \models \varphi$.

However, since $\mathfrak{T} \models \varphi_{\text{inc}}$ holds, but does not have computable Skolem functions, the games of Winter and Zimmermann are *not* a special case of the games we construct here: Our games here are still multi-player games of imperfect information (to capture the semantics of quantification) and use prophecies (for completeness), but do not require delayed moves, as prophecies can be seen as a (restricted) form of infinite lookahead. And while the existence of computable Skolem functions is concerned with bounded lookahead, here we do indeed need infinite lookahead as witnessed by the formula φ_{inc}.

Our main result shows that there is again a finite and effectively computable set of ω-regular prophecies so that the coalition of players in charge of the existentially quantified variables has a winning strategy in the game with these prophecies if and only if $\mathfrak{T} \models \varphi$. One challenge to overcome here is a careful definition of the prophecies, so that they are not leaking any information about choices for variables π_j that a player in charge of π_i with $i < j$ must not have access to.

Our result can also be framed in terms of Skolem function implementable by letter-to-letter transducers with (ω-regular) lookahead: such a transducer computes a Skolem function for an existentially quantified variable π while reading values for the variables universally quantified before π while also being able to get a regular lookahead on the trace for those universally quantified variable (much like prophecies). The use of regular lookahead is well-studied in automata theory, see e.g., [9, 11, 3].

All proofs omitted due to space restrictions can be found in the full version [25].

2 Preliminaries

For convenience, technical terms and notations in the electronic version of this manuscript are hyper-linked to their definitions (cf. `https://ctan.org/pkg/knowledge`).

Hereafter, we denote the set of nonnegative integers by $\mathbb{N}$.

Traces, Transition Systems, and Automata. An alphabet is a nonempty finite set. The sets of finite and infinite words over an alphabet Σ are denoted by Σ^* and Σ^ω, respectively. The length of finite or infinite word w is denoted by $|w| \in \mathbb{N} \cup \{\infty\}$. For a word w of length at least n, we write $w[0, n)$ for the prefix of w of length n. Given n infinite words $w_0, \ldots, w_{n-1}$, let their *merge* (also known as zip), which is an infinite word over Σ^n, be defined as

$$\mathrm{mrg}(w_0, \ldots, w_{n-1}) = (w_0(0), \ldots, w_{n-1}(0))(w_0(1), \ldots, w_{n-1}(1))(w_0(2), \ldots, w_{n-1}(2)) \cdots .$$

We define $\mathrm{mrg}(w_0, \ldots, w_{n-1})$ for finite words $w_0, \ldots, w_{n-1}$ of the same length analogously.

Let AP be a nonempty finite set of atomic propositions. A *trace* over AP is an infinite word over the alphabet 2^{AP}. Given a subset $\mathrm{AP}' \subseteq \mathrm{AP}$, the AP'-projection of a trace $t(0)t(1)t(2)\cdots$ over AP is the trace $(t(0) \cap \mathrm{AP}')(t(1) \cap \mathrm{AP}')(t(2) \cap \mathrm{AP}') \cdots \in (2^{\mathrm{AP}'})^\omega$. Now, let AP and AP' be two disjoint sets, let t be a trace over AP, and let t' be a trace over AP'. Then, we define $t \frown t'$ as the pointwise union of t and t', i.e., $t \frown t'$ is the trace over $\mathrm{AP} \cup \mathrm{AP}'$ defined as $(t(0) \cup t'(0))(t(1) \cup t'(1))(t(2) \cup t'(2)) \cdots$.

A *transition system* $\mathfrak{T} = (V, E, V_I, \lambda)$ consists of a finite set V of vertices, a set $E \subseteq V \times V$ of (directed) edges, a nonempty set $V_I \subseteq V$ of initial vertices, and a labelling $\lambda \colon V \to 2^{\mathrm{AP}}$ of the vertices by sets of atomic propositions. We assume that every vertex has at least one outgoing edge. For $v \in V$, we denote by $\mathrm{S}(v)$ the set of its successors. A *path* ρ through $\mathfrak{T}$ is an infinite sequence $\rho = v_0 v_1 v_2 \cdots$ of vertices with $v_0 \in V_I$ and $(v_n, v_{n+1}) \in E$ for every $n \geq 0$. The *trace of* ρ is defined as $\lambda(\rho) = \lambda(v_0)\lambda(v_1)\lambda(v_2)\cdots \in (2^{\mathrm{AP}})^\omega$. The *set of traces* of $\mathfrak{T}$ is $\mathrm{Tr}(\mathfrak{T}) = \{\lambda(\rho) \mid \rho \text{ is a path of } \mathfrak{T}\}$. For $V' \subseteq V$, we write $\mathfrak{T}_{V'}$ to denote the transition system (V, E, V', λ) obtained from $\mathfrak{T}$ by making V' the set of initial states, and use $\mathfrak{T}_v$ as shorthand for $\mathfrak{T}_{\{v\}}$ for $v \in V$.

A (deterministic) *parity automaton*[1] $\mathcal{A} = (Q, \Sigma, q_I, \delta, \Omega)$ consists of a finite set Q of states containing the initial state $q_I \in Q$, an alphabet Σ, a transition function $\delta \colon Q \times \Sigma \to Q$, and a coloring $\Omega \colon Q \to \mathbb{N}$ of its states by natural numbers. Let $w = w(0)w(1)w(2)\cdots \in \Sigma^\omega$. The run of $\mathcal{A}$ on w is the sequence $q_0 q_1 q_2 \cdots$ with $q_0 = q_I$ and $q_{n+1} = \delta(q_n, w(n))$ for all $n \geq 0$. A run $q_0 q_1 q_2 \cdots$ is (parity) accepting if the maximal color appearing infinitely often in the sequence $\Omega(q_0)\Omega(q_1)\Omega(q_2)\cdots$ is even. The language (parity) recognized by $\mathcal{A}$, denoted by $L(\mathcal{A})$, is the set of infinite words over Σ such that the run of $\mathcal{A}$ on w is accepting.

▶ Remark 1. Deterministic parity automata accept exactly the ω-regular languages (see, e.g., [15] for definitions).

HyperQPTL, HyperLTL and QPTL. Let $\mathcal{V}$ be a countable set of *trace variables*. The formulas of *HyperQPTL* are given by the grammar

$$\varphi ::= \exists \pi.\ \varphi \mid \forall \pi.\ \varphi \mid \psi \qquad \psi ::= \tilde{\exists} \mathtt{q}.\ \psi \mid \tilde{\forall} \mathtt{q}.\ \psi \mid \psi \mid \mathtt{p}_\pi \mid \mathtt{q} \mid \neg\psi \mid \psi \vee \psi \mid \mathbf{X}\,\psi \mid \psi\,\mathbf{U}\,\psi$$

where $\mathtt{p}$ and $\mathtt{q}$ range over AP and where π ranges over $\mathcal{V}$. Here, we use a tilde to decorate propositional quantifiers to distinguish them from trace quantifiers.

Note that there are two types of atomic formulas, i.e., propositions labeled by trace variables on which they are evaluated ($\mathtt{p}_\pi$ with $\mathtt{p} \in \mathrm{AP}$ and $\pi \in \mathcal{V}$) and unlabeled propositions ($\mathtt{q} \in \mathrm{AP}$).[2] A formula is a sentence, if every occurrence of an atomic formula $\mathtt{p}_\pi$ is in the scope of a quantifier binding π (otherwise π is said to be a free trace variable) and every occurrence of an atomic formula $\mathtt{q}$ is in the scope of a quantifier binding $\mathtt{q}$ (otherwise $\mathtt{q}$ is said to be a free propositional variable). Note that the proposition $\mathtt{p}$ in an atomic formula $\mathtt{p}_\pi$ is not considered free. Finally, we use the usual syntactic sugar like conjunction ($\wedge$), implication ($\rightarrow$), equivalence ($\leftrightarrow$), eventually ($\mathbf{F}$), and always ($\mathbf{G}$).

A (trace) *variable assignment* is a partial mapping $\Pi \colon \mathcal{V} \to (2^{\mathrm{AP}})^\omega$. Given a variable $\pi \in \mathcal{V}$, and a trace t, we denote by $\Pi[\pi \mapsto t]$ the assignment that coincides with Π on all variables but π, which is mapped to t. Let $\mathtt{q} \in \mathrm{AP}$, let $t \in (2^{\mathrm{AP}})^\omega$ be a trace over AP, and let $t_\mathtt{q} \in (2^{\{\mathtt{q}\}})^\omega$ be a trace over $\{\mathtt{q}\}$. We define the trace $t[\mathtt{q} \mapsto t_\mathtt{q}] = t' \frown t_\mathtt{q}$, where t' is the $(\mathrm{AP} \setminus \{\mathtt{q}\})$-projection of t: Intuitively, the occurrences of $\mathtt{q}$ in t are replaced according to $t_\mathtt{q}$. We lift this to sets T of traces by defining $T[\mathtt{q} \mapsto t_\mathtt{q}] = \{t[\mathtt{q} \mapsto t_\mathtt{q}] \mid t \in T\}$. Note that all traces in $T[\mathtt{q} \mapsto t_\mathtt{q}]$ have the same $\{\mathtt{q}\}$-projection, which is $t_\mathtt{q}$.

Now, for a trace assignment Π, a position $i \in \mathbb{N}$, and a nonempty set T of traces, i.e., we disregard the empty set of traces as model, we define

- $T, \Pi, i \models \mathtt{p}_\pi$ if $\mathtt{p} \in \Pi(\pi)(i)$,
- $T, \Pi, i \models \mathtt{q}$ if for all $t \in T$ we have $\mathtt{q} \in t(i)$,

[1] Note that we use parity acceptance, as we need deterministic automata for our proofs.
[2] We use different letters in these cases, but let us stress again that both $\mathtt{p}$ and $\mathtt{q}$ are propositions in AP.

- $T, \Pi, i \models \neg\psi$ if $T, \Pi, i \not\models \psi$,
- $T, \Pi, i \models \psi_1 \vee \psi_2$ if $T, \Pi, i \models \psi_1$ or $T, \Pi, i \models \psi_2$,
- $T, \Pi, i \models \mathbf{X}\,\psi$ if $T, \Pi, i+1 \models \psi$,
- $T, \Pi, i \models \psi_1\,\mathbf{U}\,\psi_2$ if there is a $j \geq i$ such that $T, \Pi, j \models \psi_2$ and $T, \Pi, j' \models \psi_1$ for all $i \leq j' < j$,
- $T, \Pi, i \models \tilde{\exists}\mathtt{q}.\ \psi$ if there exists a trace $t_{\mathtt{q}} \in (2^{\{\mathtt{q}\}})^\omega$ such that $T[\mathtt{q} \mapsto t_{\mathtt{q}}], \Pi, i \models \psi$,
- $T, \Pi, i \models \tilde{\forall}\mathtt{q}.\ \psi$ if for all traces $t_{\mathtt{q}} \in (2^{\{\mathtt{q}\}})^\omega$ we have $T[\mathtt{q} \mapsto t_{\mathtt{q}}], \Pi, i \models \psi$,
- $T, \Pi, i \models \exists\pi.\ \varphi$ if there exists a trace $t \in T$ such that $T, \Pi[\pi \mapsto t], i \models \varphi$, and
- $T, \Pi, i \models \forall\pi.\ \varphi$ if for all traces $t \in T$ we have $T, \Pi[\pi \mapsto t], i \models \varphi$.

We say that an (again nonempty) set T of traces satisfies a sentence φ, written $T \models \varphi$, if $T, \Pi_\emptyset, 0 \models \varphi$ where $\Pi_\emptyset$ is the variable assignment with empty domain. We then also say that T is a model of φ. A transition system $\mathfrak{T}$ satisfies φ, written $\mathfrak{T} \models \varphi$, if $\mathrm{Tr}(\mathfrak{T}) \models \varphi$. Let ψ be a trace quantifier-free formula that has no free (unlabeled) propositions. A labeled proposition of the form $\mathtt{p}_\pi$ in ψ is evaluated on the traces assigned to π and an unlabeled proposition of the form $\mathtt{q}$ is evaluated on a trace "selected" by the quantifier binding $\mathtt{q}$. Hence, $T, \Pi, i \models \psi$ is independent of T and we often write $\Pi \models \psi$ instead of $T, \Pi, 0 \models \psi$.

Some comments on the definition of HyperQPTL are due.

▶ Remark 2. We have, for technical reasons, defined HyperQPTL so that every formula has a prefix of trace quantifiers followed by a formula (possibly) containing propositional quantifiers, but no more trace quantifiers. On the other hand, Finkbeiner et al. [13] require formulas to be in prenex normal form, but allow to mix trace and propositional quantification in the quantifier prefix. We refrained from doing so, as one can duplicate the vertices of the transition system one is interested in, so that it has enough paths to simulate propositional quantification by trace quantification, and can adapt the formula correspondingly. This construction has a linear blowup.

▶ Remark 3. *HyperLTL* is the fragment of HyperQPTL sentences that do not use the propositional quantifiers $\tilde{\exists}$ and $\tilde{\forall}$ (and thus also not use unlabeled propositions).

We say that a HyperQPTL formula φ is a *QPTL* sentence if it has no trace quantifiers and no free propositional variables, i.e., all unlabeled propositional variables are in the scope of a propositional quantifier. But a QPTL sentence may have free trace variables, say $\pi_0, \ldots, \pi_{k-1}$ for some $k \geq 0$. Typically, QPTL is defined without trace variables labelling propositions and QPTL formulas define languages over the alphabet 2^{AP} [23]. For technical necessity, we allow such labels, which implies that our formulas define languages over the alphabet $(2^{\mathrm{AP}})^k$, where k is the number of trace variables occurring in the formula. More formally, a QPTL sentence with trace variables $\pi_0, \pi_1, \ldots, \pi_{k-1}$ defines the language

$$L(\varphi) = \{\mathrm{mrg}(t_0, \ldots, t_{k-1}) \mid t_0, \ldots, t_{k-1} \in (2^{\mathrm{AP}})^\omega : \Pi_\emptyset[\pi_0 \mapsto t_0, \ldots, \pi_{k-1} \mapsto t_{k-1}] \models \varphi\}.$$

Now, let $\varphi = Q_0\pi_0 Q_1\pi_1 \cdots Q_{k-1}\pi_{k-1}.\ \psi$ with $Q_i \in \{\exists, \forall\}$ and trace quantifier-free ψ be a HyperQPTL sentence and define $\varphi_i = Q_{i+1}\pi_{i+1}Q_{i+2}\pi_{i+2} \cdots Q_{k-1}\pi_{k-1}.\ \psi$ for $i \in \{0, 1, \ldots, k-1\}$ and $\varphi_{-1} = \varphi$. Note that $\varphi_{k-1} = \psi$ and that the free trace variables of each φ_i with $i \in \{-1, 0, \ldots, k-1\}$ are exactly $\pi_0, \ldots, \pi_i$. The following results follows by combining and adapting automata constructions for classical QPTL and HyperLTL [6, 14, 23].

▶ **Proposition 4.**

1. *Let $\mathfrak{T}$ be a transition system. For every $i \in \{-1, 0, \dots, k-1\}$ there is an (effectively constructible) parity automaton $\mathcal{A}_i^{\mathfrak{T}}$ such that*

$$L(\mathcal{A}_i^{\mathfrak{T}}) = \{\mathrm{mrg}(\Pi(\pi_0), \dots, \Pi(\pi_i)) \mid \Pi(\pi_j) \in \mathrm{Tr}(\mathfrak{T}) \text{ for } 0 \leq j \leq i \text{ and } \mathrm{Tr}(\mathfrak{T}), \Pi, 0 \models \varphi_i\}.$$

2. *A language over $(2^{\mathrm{AP}})^k$ is ω-regular if and only if it is of the form $L(\varphi)$ for a QPTL-sentence φ over some $\mathrm{AP}' \supseteq \mathrm{AP}$ with k free trace variables.*[3]

3 Gamed-based Model-Checking for HyperQPTL

In this section, we present our game-based characterization of $\mathfrak{T} \models \varphi$ for finite transition systems $\mathfrak{T}$ and HyperQPTL sentences φ. To this end, we introduce multi-player games with hierarchical information in Section 3.1 and then present the construction of our game in Section 3.2, while we introduce prophecies and prove their soundness in Section 3.3. Then, in Section 4, we show how to construct complete prophecies for the special case of safety formulas (to be defined formally there). Finally, in Section 5, we show how to construct complete prophecies for arbitrary formulas. This allows us to first explain how to generalize the prophecy definition from $\forall^*\exists^*$ to arbitrary quantifier prefixes and then, in a second step, move from safety to ω-regular languages.

3.1 Multi-player Graph Games with Hierarchical Information

We develop a game-based characterization of model-checking for HyperQPTL via (multi-player) graph games with hierarchical information, using the notations of Berwanger et al. [4]. First, we introduce the necessary definitions and then present our game in Section 3.2. The games considered by Berwanger et al. are concurrent games (i.e., the players make their moves simultaneously), while for our purpose turn-based games (i.e., the players make their moves one after the other) are sufficient. Turn-based games are simpler versions of concurrent games. To avoid cumbersome notation, we introduce a turn-based variant of these games.

Fix some finite set C of players forming a coalition playing against a distinguished agent called Nature (which is *not* in C). For each player $i \in C$ we fix a finite set B^i of observations. A game graph $G = (V, E, v_I, (\beta^i)_{i \in C})$ consists of a finite set $V = \biguplus_{i \in C} V_i \uplus V_{\mathrm{Nat}}$ of positions partitioned into sets controlled by some player resp. Nature, an edge relation $E \subseteq V \times V$ representing moves, an initial position $v_I \in V$, and a collection $(\beta^i)_{i \in C}$ of observation functions $\beta^i \colon V \to B^i$ that label, for each player, the positions with observations. We require that E has no dead-ends, i.e., for every $v \in V$ there is a $v' \in V$ with $(v, v') \in E$.

A game graph $(V, E, v_I, (\beta^i)_{i \in C})$ yields hierarchical information if there exists a total order $\preceq$ over C such that if $i \preceq j$ then for all $v, v' \in V$, $\beta^i(v) = \beta^i(v')$ implies $\beta^j(v) = \beta^j(v')$, i.e., if Player i cannot distinguish v and v', then neither can Player j for $i \preceq j$.

Intuitively, a play starts at position $v_I \in V$. At position v, the player that controls this position chooses a successor position v' such that $(v, v') \in E$. Now, each player $i \in C$ receives the observation $\beta^i(v')$ and the play continues from position v'. Thus, a play of G is an infinite sequence $v_0 v_1 v_2 \cdots$ of vertices such that $v_0 = v_I$ and for all $r \geq 0$ we have $(v_r, v_{r+1}) \in E$.

[3] For the sake of readability, in the following, we do not distinguish between AP and AP′, i.e., we assume that AP always contains enough propositions not used in our languages to quantify over.

A history is a prefix $v_0 v_1 \cdots v_r$ of a play. We denote the set of all histories by $\mathrm{Hist}(G)$ and extend $\beta^i \colon V \to B^i$ to plays and histories by defining $\beta^i(v_0 v_1 v_2 \cdots) = \beta^i(v_1)\beta^i(v_2)\beta^i(v_3)\cdots$. Note that the observation of the initial position is discarded for technical reasons [4]. We say two histories h and h' are indistinguishable to Player $i \in C$, denoted by $h \sim_i h'$, if $\beta^i(h) = \beta^i(h')$.

A strategy for Player $i \in C$ is a mapping $s^i \colon V^* \to V$ that satisfies $s^i(h) = s^i(h')$ for all histories h, h' with $h \sim_i h'$ (i.e., the move selected by the strategy only depends on the observations of the history). A play $v_0 v_1 v_2 \cdots$ is consistent with s^i if for every $r \geq 0$, we have $v_{r+1} = s^i(v_0 v_1 \cdots v_r)$. A play is consistent with a collection of strategies $(s^i)_{i \in C}$ if it is consistent with each s^i. The set of possible outcomes of a collection of strategies is the set of all plays that are consistent with it. As usual, a strategy is finite-state, if it is implemented by some Moore machine.

Lastly, a game $\mathcal{G}$ consists of a game graph G and a winning condition $W \subseteq V^\omega$, where V is the set of positions of G. A play is winning if it is in W and a collection of strategies is winning if all its outcomes are winning.

▶ **Proposition 5** ([21, 4]).

1. *It is decidable, given a game with hierarchical information with ω-regular winning condition, whether it has a winning collection of strategies.*
2. *A game with hierarchical information with ω-regular winning condition has a winning collection of strategies if and only if it has a winning collection of finite-state strategies.*

3.2 The Model-checking Game

For the remainder of this paper, we fix a HyperQPTL sentence φ and a transition system $\mathfrak{T}$. We assume (w.l.o.g.)[4]

$$\varphi = \forall \pi_0 \exists \pi_1 \cdots \forall \pi_{k-2} \exists \pi_{k-1}.\ \psi$$

such that ψ is trace quantifier-free, define

$$\varphi_i = Q_{i+1}\pi_{i+1} Q_{i+2}\pi_{i+2} \cdots \forall \pi_{k-2} \exists \pi_{k-1}.\ \psi$$

for $i \in \{-1, 0, \ldots, k-1\}$ and use the automata $\mathcal{A}_i^{\mathfrak{T}}$ constructed in Proposition 4 satisfying

$$L(\mathcal{A}_i^{\mathfrak{T}}) = \{\mathrm{mrg}(\Pi(\pi_0), \ldots, \Pi(\pi_i)) \mid \Pi(\pi_j) \in \mathrm{Tr}(\mathfrak{T}) \text{ for all } 0 \leq j \leq i \text{ and } \mathrm{Tr}(\mathfrak{T}), \Pi, 0 \models \varphi_i\}.$$

We use the notation $(\mathcal{A}_i^{\mathfrak{T}})_q$ to denote the parity automaton obtained from $\mathcal{A}_i^{\mathfrak{T}}$ by making its state q the initial state. Finally, let $\mathcal{A}_{k-1}^{\mathfrak{T}} = (Q, \Sigma, q_I, \delta, \Omega)$. Note that $\mathcal{A}_{k-1}^{\mathfrak{T}}$ accepts the trace assignments coming from paths trough $\mathfrak{T}$ that satisfy ψ. We say that $\mathcal{A}_{k-1}^{\mathfrak{T}}$ checks ψ.

We define a multi-player game $\mathcal{G}(\mathfrak{T}, \varphi)$ with hierarchical information induced by the transition system $\mathfrak{T}$ and the HyperQPTL sentence φ. This game is played between *Falsifier* (who takes on the role of Nature, cf. Section 3.1), who is in charge of providing traces (from paths trough the transition system) for the universally quantified variables, and a coalition of *Verifier-players* $\{1, 3, \ldots, k-1\}$ (*Verifier i* for short), who are in charge of providing traces (also from paths trough the transition system) for the existentially quantified variables. The goal of Falsifier is to prove $\mathfrak{T} \not\models \varphi$ and the goal of the coalition of Verifier-players is to prove

[4] The following reasoning can easily be extended to general sentences with arbitrary quantifier prefixes, albeit at the cost of more complex notation. We substantiate this claim in Remark 20 on Page 16.

$\mathfrak{T} \models \varphi$. Therefore, the k traces built during a play are read synchronously by the parity automaton $\mathcal{A}^{\mathfrak{T}}_{k-1}$ accepting the trace assignments that satisfy ψ (recall that ψ is the trace quantifier-free part of φ).

The game has two phases, an initialization phase where initial vertices for all paths through the transition system are picked, and a second phase (of infinite duration) where the paths (which induce the trace assignment) are built. Formally, in the initialization phase, a position of the game is of the form $((v_0, \dots, v_{i-1}, \underbrace{\bullet, \dots, \bullet}_{k-i \text{ times}}), q_I, i)$ where $v_0, \dots, v_{i-1} \in V_I$ are initial vertices in the transition system $\mathfrak{T}$, $\bullet$ is a fresh (placeholder) symbol, $q_I \in Q$ is the initial state of $\mathcal{A}^{\mathfrak{T}}_{k-1}$, and $i \in \{0, 1, \cdots, k-1\}$. In the second phase, a position of the game is of the form $((v_0, \dots, v_{k-1}), q, i)$ where $v_0, v_1, \dots, v_{k-1} \in V$, $q \in Q$, and $i \in \{0, 1, \dots, k-1\}$. A vertex whose last component is an odd i is controlled by Verifier i and a vertex whose last component is even is controlled by Falsifier.

The edges (also called moves) of the game graph are defined as follows, where the first two items are the moves in the initialization phase:

- $\big(((v_0, \dots, v_{i-1}, \bullet, \dots, \bullet), q_I, i), ((v_0, \dots, v_{i-1}, v_i, \bullet, \dots, \bullet), q_I, i+1)\big)$ for all $v_i \in V_I$ and all $i \in \{0, 1, \dots, k-2\}$: An initial vertex in $\mathfrak{T}$ for the i-th path is picked.
- $\big(((v_0, \dots, v_{k-2}, \bullet), q_I, k-1), ((v_0, \dots, v_{k-2}, v_{k-1}), q, 0)\big)$ for all $v_{k-1} \in V_I$ where $q = \delta(q_I, \mathrm{mrg}(\lambda(v_0), \dots, \lambda(v_{k-1})))$: An initial vertex in $\mathfrak{T}$ for the last path is picked. With this move, the initialization phase is over and the state of $\mathcal{A}^{\mathfrak{T}}_{k-1}$ checking ψ is updated for the first time.
- $\big(((v'_0, \dots, v'_{i-1}, v_i, v_{i+1}, \dots, v_{k-1}), q, i), ((v'_0, \dots, v'_{i-1}, v'_i, v_{i+1}, \dots, v_{k-1}), q, i+1)\big)$ for all $(v_i, v'_i) \in E$ and all $i \in \{0, 1, \dots, k-2\}$: The i-th path is updated by moving to a successor vertex in $\mathfrak{T}$.
- $\big(((v'_0, \dots, v'_{k-2}, v_{k-1}), q, k-1), ((v'_0, \dots, v'_{k-2}, v'_{k-1}), q', 0)\big)$ for all $(v_{k-1}, v'_{k-1}) \in E$ where $q' = \delta(q, \mathrm{mrg}(\lambda(v'_0), \lambda(v'_1), \dots, \lambda(v'_{k-1})))$: The last path is updated by moving to a successor vertex in $\mathfrak{T}$. Simultaneously, the state of $\mathcal{A}^{\mathfrak{T}}_{k-1}$ checking ψ is updated.

The initial position is $((\bullet, \dots, \bullet), q_I, 0)$. We use a parity winning condition. The color of all positions of the initialization phase is 0 (the color is of no consequence as these vertices are seen only once during the course of a play). The color of positions of the form $((v_0, \dots, v_{k-1}), q, i)$ is the color that q has in $\mathcal{A}^{\mathfrak{T}}_{k-1}$, i.e., a play is winning for the Verifier-players if and only if the trace assignment picked by them and Falsifier satisfies ψ.

The game described must be a game of hierarchical information to capture the fact that the Skolem function for an existentially quantified π_i depends only on the universally quantified variables π_j with $j \in \{0, 2, \dots, i-1\}$. To capture that, we define an equivalence relation $\equiv_i$ between positions of the game for $i \in \{1, 3, \dots, k-1\}$ which ensures that for Verifier i, two positions are indistinguishable if they coincide on their first $i+1$ components and additionally belong to the same player. Formally, regarding positions from the initialization phase, let $((v_0, \dots, v_{m-1}, \bullet, \dots, \bullet), q_I, m)$ and $((v'_0, \dots, v'_{n-1}, \bullet, \dots, \bullet), q_I, n)$ be $\equiv_i$-equivalent if $v_j = v'_j$ for all $j \leq i$ and $m = n$. Regarding all other positions, let $((v_0, \dots, v_{k-1}), p, m)$ and $((v'_0, \dots, v'_{k-1}), q, n)$ be $\equiv_i$-equivalent if $v_j = v'_j$ for all $j \leq i$ and $m = n$. Now, we define the observation functions: For Verifier i, it maps positions to their $\equiv_i$-equivalence classes.

3.3 The Model-Checking Game with Prophecies

The game $\mathcal{G}(\mathfrak{T}, \varphi)$ as introduced in Section 3.2 does not capture $\mathfrak{T} \models \varphi$. While it is sound (see Lemma 8 for the empty set of prophecies), i.e., the coalition of Verifier-players having a winning collection of strategies for $\mathcal{G}(\mathfrak{T}, \varphi)$ implies that $\mathfrak{T} \models \varphi$, but the converse is not necessarily true. This is witnessed, e.g., by a transition system $\mathfrak{T}$ with $\mathrm{Tr}(\mathfrak{T}) = (2^{\{\mathtt{p}\}})^\omega$ and

the sentence $\forall\pi.\ \exists\pi'.\ \mathtt{p}_{\pi'} \leftrightarrow \mathbf{F}\,\mathtt{p}_{\pi}$. As explained in the introduction, the Verifier-player does not have a strategy to select, step-by-step, a trace t' for π' while given, again step-by-step, a trace t for π, as the choice of the first letter of t' depends on all positions of t. Hence, the coalition does not win $\mathcal{G}(\mathfrak{T}, \varphi)$, even though $\mathfrak{T} \models \varphi$.

In the following, we show how *prophecies*, binding commitments about future moves by Falsifier, make the game-based approach to model-checking complete. In our example, Falsifier has to make, with his first move, a commitment about whether he will ever play a $\mathtt{p}$ or not. This allows the Verifier-player to pick the "right" first letter of t' and thereby win the game, if Falsifier honors the commitment. If not, the rules of the game make him loose. To add prophecies to $\mathcal{G}(\mathfrak{T}, \varphi)$, we do not change the rules of the game, but instead modify the transition system the game is played on (to allow Falsifier to select truth values for the *prophecy variables*[5], which is the mechanism he uses to make the commitments) and modify the formula (to ensure that Falsifier loses if he breaks a commitment). Hence, given $\mathfrak{T}$ and φ, we construct $\mathfrak{T}^{\mathcal{P}}$ and $\varphi^{\mathcal{P},\Xi}$ such that the following two properties are satisfied:

- Soundness: If the coalition of Verifier-players has a winning collection of strategies for $\mathcal{G}(\mathfrak{T}^{\mathcal{P}}, \varphi^{\mathcal{P},\Xi})$, then $\mathfrak{T} \models \varphi$.
- Completeness: If $\mathfrak{T} \models \varphi$, then the coalition of Verifier-players has a winning collection of strategies for $\mathcal{G}(\mathfrak{T}^{\mathcal{P}}, \varphi^{\mathcal{P},\Xi})$.

The challenge here is to construct the "right" prophecies that allow us to prove completeness, as soundness is independent of the prophecies chosen.

Recall that $\pi_0, \pi_2, \ldots, \pi_{k-2}$ resp. $\pi_1, \pi_3, \ldots, \pi_{k-1}$ are the universally resp. existentially quantified variables in our fixed formula φ. We frequently need to refer to their indices. Hence, we define $I_\forall = \{0, 2, \ldots, k-2\}$ and $I_\exists = \{1, 3, \ldots, k-1\}$.

We begin by defining the transition system $\mathfrak{T}^{\mathcal{P}}$ from the fixed transition system $\mathfrak{T}$, in which paths additionally determine truth values for prophecy variables.

▶ **Definition 6** (*System manipulation*). *Let $\mathcal{P} = (\mathrm{P}_i)_{i \in I_\forall}$ be a collection of sets of atomic propositions such that* AP*, the P_i, and $\{\mathtt{m}_i \mid i \in I_\forall\}$ are all pairwise disjoint, where the $\mathtt{m}_i$ are propositions used to mark copies of the transition system $\mathfrak{T} = (V, E, V_I, \lambda)$.*

For $i \in I_\forall$, we define $\mathfrak{T}^{\mathrm{P}_i} = (V^{\mathrm{P}_i}, E^{\mathrm{P}_i}, V_I^{\mathrm{P}_i}, \lambda^{\mathrm{P}_i})$ over $\mathrm{AP} \uplus \mathrm{P}_i \uplus \{\mathtt{m}_i\}$ where $V^{\mathrm{P}_i} = V \times 2^{\mathrm{P}_i} \times \{i\}$, $E^{\mathrm{P}_i} = \{((s, A, i), (s', A', i)) \mid (s, s') \in E \text{ and } A, A' \in 2^{\mathrm{P}_i}\}$, $V_I^{\mathrm{P}_i} = V_I \times 2^{\mathrm{P}_i} \times \{i\}$ and $\lambda^{\mathrm{P}_i}(s, A, i) = \lambda(s) \cup A \cup \{\mathtt{m}_i\}$.

Furthermore, we define $\mathfrak{T}^{\mathcal{P}} = (V^{\mathcal{P}}, E^{\mathcal{P}}, V_I^{\mathcal{P}}, \lambda^{\mathcal{P}})$ as the disjoint union of the $\mathfrak{T}^{\mathrm{P}_i}$, where a vertex of $\mathfrak{T}^{\mathcal{P}}$ is in $V_I^{\mathcal{P}}$ if and only if it is in some $V_I^{\mathrm{P}_i}$. Consequently, we have $\mathrm{Tr}(\mathfrak{T}^{\mathcal{P}}) = \bigcup_{i \in I_\forall} \{t \frown t' \frown \{\mathtt{m}_i\}^\omega \mid t \in \mathrm{Tr}(\mathfrak{T}) \text{ and } t' \in (2^{\mathrm{P}_i})^\omega\}$.

An effect of this definition is that all paths trough the manipulated system select truth values for the prophecy variables with each move. Furthermore, each such path is marked by a proposition of the form $\mathtt{m}_i$ indicating for which i the prophecy variables are selected. This will later be used to ensure that Falsifier selects the correct prophecies (cf. Definition 7) for each path he picks for a universally quantified variable. For the Verifier-players, this additional information is simply ignored, as can be seen from the manipulated formula (cf. Definition 7) we introduce now.

We define the sentence $\varphi^{\mathcal{P},\Xi}$ which ensures that Falsifier loses, if he breaks his commitments made via prophecy variables: for every prophecy variable, we have an associated *prophecy*, a language $\mathfrak{P} \subseteq ((2^{\mathrm{AP}})^i)^\omega$ for some i. Note that Falsifier can only make commitments about

[5] Note that prophecy variables are Boolean variables that are set by Falsifier during each move of a play and should not be confused with trace variables or with propositional variables.

his own moves, as the Verifier-players could otherwise falsify the commitments (made by Falsifier) about their moves: Prophecies can only refer to traces for universally quantified variables. Also, we will have prophecies for each universally quantified variable.

▶ **Definition 7** (*Property manipulation*). *Let $\Xi = (\Xi_i)_{i\in I_\forall}$ be a family $\Xi_i = \{\xi_{i,1}, \ldots, \xi_{i,n_i}\}$ of sets of QPTL sentences (over* AP*) such that each $\xi \in \Xi_i$ uses only trace variables π_j with even $j \leq i$. Let $\mathcal{P} = (\mathrm{P}_i)_{i\in I_\forall}$ satisfy the disjointness requirements of Definition 6 and $\mathrm{P}_i = \{\mathtt{p}_{i,1}, \ldots, \mathtt{p}_{i,n_i}\}$, i.e., we have $|\Xi_i| = |\mathrm{P}_i|$. We define the HyperQPTL sentence $\varphi^{\mathcal{P},\Xi}$ as*

$$\forall\pi_0\exists\pi_1\cdots\forall\pi_{k-2}\exists\pi_{k-1}.\ \left[\bigwedge\nolimits_{i\in I_\forall}(\mathtt{m}_i)_{\pi_i} \wedge \mathbf{G}\left(\bigwedge\nolimits_{\ell=1}^{n_i}((\mathtt{p}_{i,\ell})_{\pi_i} \leftrightarrow \xi_{i,\ell})\right)\right] \rightarrow \psi.$$

We denote the trace quantifier-free part of $\varphi^{\mathcal{P},\Xi}$ by $\psi^{\mathcal{P},\Xi}$.

Note that $\varphi^{\mathcal{P},\Xi}$ has the same quantifier prefix as φ and that restricting each $\xi_{i,\ell}$ to trace variables π_j with even $j \leq i$ ensures that the prophecies only refer to trace variables under the control of Falsifier. Also note that the truth values of prophecy variables on trace variables under the control of the Verifier-players are not used in the formula. Finally, Falsifier has to pick the i-th path in $\mathfrak{T}^{\mathrm{P}_i}$ (and thus select valuations for the prophecy variables in P_i), otherwise he loses immediately.

Our construction is sound, independently of the choice of prophecies.

▶ **Lemma 8.** *Let Ξ and $\mathcal{P}$ satisfy the requirements of Definition 7. If the coalition of Verifier-players has a winning collection of strategies in $\mathcal{G}(\mathfrak{T}^{\mathcal{P}}, \varphi^{\mathcal{P},\Xi})$, then $\mathfrak{T} \models \varphi$.*

4 Complete Prophecies for Safety Properties

We show that there are sets Ξ and $\mathcal{P}$ such that if $\mathfrak{T} \models \varphi$, then the coalition of Verifier-players wins $\mathcal{G}(\mathfrak{T}^{\mathcal{P}}, \varphi^{\mathcal{P},\Xi})$. We first consider the case where ψ is a safety property, i.e., the automaton $\mathcal{A}^{\mathfrak{T}}_{k-1}$, which checks ψ, is a deterministic safety automaton. A *safety automaton* is a parity automaton using only two colors, an even one and an odd one, and moreover, all its states are even-colored except for an odd-colored sink state. In other words, all runs avoiding the single unsafe state are accepting. We start with safety as this allows us to work with "simpler" prophecies while presenting all underlying concepts needed for arbitrary quantifier prefixes. The idea for safety is that a prophecy should indicate which successor vertices are safe for the Verifier-players to move to, i.e., from which successor vertices it is possible to successfully continue the play without immediately losing. In the general case, we have to additionally handle a more complex acceptance condition. This is shown in Section 5.

▶ **Definition 9** (Prophecy construction for safety). *For each $i \in I_\forall$, each vector $\bar{v} = (v_0, v_1, \ldots, v_{i+1})$ of vertices of $\mathfrak{T}$, and each state q of $\mathcal{A}^{\mathfrak{T}}_{i+1}$ we define*

$$(\mathfrak{S}_i)^{\bar{v}}_q = \{\mathrm{mrg}(t_0, t_2, \ldots, t_i) \mid t_0 \in \mathrm{Tr}(\mathfrak{T}_{v_0}), t_2 \in \mathrm{Tr}(\mathfrak{T}_{v_2}), \ldots, t_i \in \mathrm{Tr}(\mathfrak{T}_{v_i}),\ \text{and there are}$$
$$t_1 \in \mathrm{Tr}(\mathfrak{T}_{v_1}), t_3 \in \mathrm{Tr}(\mathfrak{T}_{v_3}), \ldots, t_{i+1} \in \mathrm{Tr}(\mathfrak{T}_{v_{i+1}})\ \text{s.t.}\ \mathrm{mrg}(t_0, \ldots, t_{i+1}) \in L((\mathcal{A}^{\mathfrak{T}}_{i+1})_q)\}.$$

Note that the t_i for $i \in I_\exists$ may depend on all t_j with $j \in I_\forall$. Our game definition ensures that the choice for an existential variable π_i only depends on the choices for π_j with $j < i$.

Also note that each prophecy is an ω-regular language, as ω-regular languages are closed under projection, the analogue of existential quantification. So, due to Proposition 4, there are QPTL-sentences expressing the prophecies allowing us to verify them in the formula $\varphi^{\mathcal{P},\Xi}$.

Next, we define the sets $\Xi = (\Xi_i)_{i\in I_\forall}$ and $\mathcal{P} = (\mathrm{P}_i)_{i\in I_\forall}$ of QPTL formulas expressing the prophecies defined above and the corresponding prophecy variables.

▶ **Definition 10.** *Let Ξ_i be the set of QPTL formulas that contains sentences $(\xi_i)_q^{\bar{v}}$ for each state q of $\mathcal{A}_{i+1}^{\mathfrak{T}}$ and each vector $\bar{v}$ of vertices expressing the prophecy $(\mathfrak{S}_i)_q^{\bar{v}}$ using only trace variables π_j with even $j \leq i$. Let P_i be the set that contains the prophecy variable $(\mathfrak{p}_i)_q^{\bar{v}}$ for each state q of $\mathcal{A}_{i+1}^{\mathfrak{T}}$ and each vector $\bar{v}$ of vertices. The prophecy variable $(\mathfrak{p}_i)_q^{\bar{v}}$ corresponds to $(\xi_i)_q^{\bar{v}}$.*

Recall that ψ is the maximal trace quantifier-free subformula of φ. Our main technical lemma shows that the prophecies defined above are indeed complete.

▶ **Lemma 11.** *Let $\mathfrak{T} \models \varphi$ such that ψ is a safety property. Then the coalition of Verifier-players has a winning collection of strategies in $\mathcal{G}(\mathfrak{T}^{\mathcal{P}}, \varphi^{\mathcal{P},\Xi})$ with Ξ and $\mathcal{P}$ as in Definition 10.*

Before we turn to the proof of Lemma 11, we introduce some notation about the game $\mathcal{G}(\mathfrak{T}^{\mathcal{P}}, \varphi^{\mathcal{P},\Xi})$ and define the strategies we will prove winning.

Firstly, recall that $\mathfrak{T}^{\mathcal{P}}$ is the union $\biguplus_{i \in I_\forall} \mathfrak{T}^{P_i}$. Thus, a vertex v of $\mathfrak{T}^{\mathcal{P}}$ is of the form $(s, A, i) \in V \times 2^{\mathrm{P}_i} \times \{i\}$ for some P_i, where V is the set of vertices of $\mathfrak{T}$, and the label $\lambda^{\mathrm{P}_i}(v)$ of v is $\lambda(s) \cup A \cup \{\mathtt{m}_i\}$, i.e., the union of its original (meaning AP-based) label $\lambda(s)$ in $\mathfrak{T}$, its associated prophecy variables A, and its marker $\mathtt{m}_i$ indicating that v belongs to $\mathfrak{T}^{\mathrm{P}_i}$. We need to access these bits of information. Hence, we define $\lambda_{\mathrm{AP}}(v) = \lambda(s)$, $\mathrm{Prophecies}(v) = A$, and $\mathrm{mark}(v) = \mathtt{m}_i$. Given a path $\rho = v_0 v_1 v_2 \cdots$ through $\mathfrak{T}^{\mathcal{P}}$, let $\lambda_{\mathrm{AP}}(\rho) = \lambda_{\mathrm{AP}}(v_0)\lambda_{\mathrm{AP}}(v_1)\lambda_{\mathrm{AP}}(v_2)\cdots$. Moreover, for $v = (s, A, i)$ in $\mathfrak{T}^{\mathrm{P}_i}$, we are often interested in reasoning about s in $\mathfrak{T}$. To simplify our notation, we will also write v for the (unique) vertex s in $\mathfrak{T}$ induced by v.

Secondly, recall that $\psi^{\mathcal{P},\Xi}$ is the trace quantifier-free part of $\varphi^{\mathcal{P},\Xi}$. Let $\mathcal{B}$ denote the parity automaton $(Q_{\mathcal{B}}, \Sigma, q_I^{\mathcal{B}}, \delta_{\mathcal{B}}, \Omega_{\mathcal{B}})$ that accepts the trace assignments that satisfy $\psi^{\mathcal{P},\Xi}$.

Lastly, recall that a position of $\mathcal{G}(\mathfrak{T}^{\mathcal{P}}, \varphi^{\mathcal{P},\Xi})$ is of the form $((v'_0, \ldots, v'_{i-1}, v_i, \ldots, v_{k-1}), q, i)$ where $v'_0, \ldots, v'_{i-1}, v_i, \ldots, v_{k-1}$ are vertices in the transition system $\mathfrak{T}^{\mathcal{P}}$, $q \in Q_{\mathcal{B}}$ is a state of the parity automaton $\mathcal{B}$ checking $\psi^{\mathcal{P},\Xi}$, and $i \in \{0, 1, \ldots, k-1\}$. Technically, in the initialization phase, a position is of the form $((v_0, \ldots, v_{i-1}, \underbrace{\bullet, \ldots, \bullet}_{k-i \text{ times}}), q_I, i)$ where $v_0, \ldots, v_{i-1}$ are vertices and $\bullet$ is a placeholder. For simplicity, we always refer to a position with $((v'_0, \ldots, v'_{i-1}, v_i, \ldots, v_{k-1}), q, i)$, even though $\bullet$ might be present. Also, for convenience, we define the successors $\mathrm{S}(\bullet)$ of the placeholder symbol $\bullet$ to be V_I, the set of initial vertices of $\mathfrak{T}$.

We say a play in $\mathcal{G}(\mathfrak{T}^{\mathcal{P}}, \varphi^{\mathcal{P},\Xi})$ is in *Round* (r, i) for $r \geq 0$ and $i \in \{0, 1, \ldots, k-1\}$ if the play is in a position of the form $((v'_0, \ldots, v'_{i-1}, v_i, \ldots, v_{k-1}), q, i)$ for the $(r+1)$-th time. Also, we say we are in *Round* r if the play is in Round (r, i) for some i.

Given a play α in $\mathcal{G}(\mathfrak{T}^{\mathcal{P}}, \varphi^{\mathcal{P},\Xi})$, for $i \in \{0, 1, \ldots, k-1\}$, let ρ_i^α denote the path through $\mathfrak{T}^{\mathcal{P}}$ that is induced by α when considering the moves made in Round (r, i) for $r \geq 0$: Formally, if in Round (r, i) the move from position $p = ((v'_0, \ldots, v'_{i-1}, v_i, \ldots, v_{k-1}), q, i)$ to $p' = ((v'_0, \ldots, v'_{i-1}, v'_i, \ldots, v_{k-1}), q', (i+1) \mod k)$ is made, then $\rho_i^\alpha(r) = v'_i$. Note that $q' = q$, unless $i = k-1$, then $q' = \delta_{\mathcal{B}}(q, \mathrm{mrg}(\lambda^{\mathcal{P}}(v'_0), \ldots, \lambda^{\mathcal{P}}(v'_{k-1})))$. Furthermore, note that the move (p, p') is uniquely identified by p and v'_i. So in the future, we simply write the move from v_i to v'_i when p is clear from the context. We drop the index α from ρ_i^α and write ρ_i when α is clear from the context.

We continue by defining a collection of strategies for the Verifier-players and then show that $\mathfrak{T} \models \varphi$ implies that this collection is winning in $\mathcal{G}(\mathfrak{T}^{\mathcal{P}}, \varphi^{\mathcal{P},\Xi})$. Recall that $\varphi^{\mathcal{P},\Xi}$ is $\forall\pi_0 \exists\pi_1 \cdots \forall\pi_{k-2} \exists\pi_{k-1}.\ \psi^{\mathcal{P},\Xi}$. Assume the play is in Round (r, i) for $r \geq 0$ and $i \in I_\exists$. Thus, it is in a position of the form

$$((v'_0, \ldots, v'_{i-1}, v_i, \ldots, v_{k-1}), q, i)$$

and Verifier i has to move. We review some information Verifier i can use to base her choice on. In Round r, Verifier i has access to the first r letters of the traces $\lambda_{\mathrm{AP}}(\rho_0), \lambda_{\mathrm{AP}}(\rho_1), \ldots, \lambda_{\mathrm{AP}}(\rho_i)$ induced by the play.[6] Let q_i be the state that $\mathcal{A}_i^{\mathfrak{T}}$ has reached on $\mathrm{mrg}(\lambda_{\mathrm{AP}}(\rho_0)[0,r), \lambda_{\mathrm{AP}}(\rho_1)[0,r), \ldots, \lambda_{\mathrm{AP}}(\rho_i)[0,r))$ for $i \in I_\exists$.

▶ **Definition 12** (Strategy definition for safety). *Let*

$$M_i(v_0', \ldots, v_{i-1}', v_i, q_i) = \{v_i' \in \mathrm{S}(v_i) \mid (\mathfrak{p}_{i-1})_{q_i}^{\bar{v}} \in \mathrm{Prophecies}(v_{i-1}')\}$$

where $\bar{v} = (v_0', \ldots, v_i')$. *If* $M_i(v_0', \ldots, v_{i-1}', v_i, q_i)$ *is nonempty, then Verifier* i *can move to any* v_i' *in the set (the "*Nonempty*-case"). If it is empty, then Verifier* i *can move to any* $v_i' \in \mathrm{S}(v_i)$ *(the "*Empty*-case").*

Proof sketch of Lemma 11. We show that the just constructed strategies form a winning collection for the Verifier-players. Here, we reason about a play under the assumption that all prophecies are set correctly (i.e., the premise of $\psi^{\mathcal{P},\Xi}$ is true) as otherwise, the play is trivially won by the Verifier-players. Under this assumption, and the basis that $\mathfrak{T} \models \varphi$, we prove that the Verifier-players have always used the Nonempty-case to pick their moves. This implies that the play is winning for them: We have to argue that the conclusion of $\psi^{\mathcal{P},\Xi}$ (which is ψ) is true. Recall that $\mathcal{A}_{k-1}^{\mathfrak{T}}$ checks ψ. Let $t_0, \ldots, t_{k-1}$ be the traces induced by the play. We show that $\mathcal{A}_{k-1}^{\mathfrak{T}}$ accepts $\mathrm{mrg}(t_0, \ldots, t_{k-1})$. In Round $(r, k-1)$, $\mathcal{A}_{k-1}^{\mathfrak{T}}$ has reached a state q_{k-1} on the prefix of length r of $\mathrm{mrg}(t_0, \ldots, t_{k-1})$, the prophecy $(\mathfrak{S}_{k-1})_{q_{k-1}}^{\bar{v}}$ for the correct $\bar{v}$ (cf. Definition 12) holds, which implies that $L((\mathcal{A}_{k-1}^{\mathfrak{T}})_{q_{k-1}}) \neq \emptyset$. Hence, q_{k-1} is a safe state, and not the single (unsafe) sink state. Thus, an induction shows that the run of $\mathcal{A}_{k-1}^{\mathfrak{T}}$ on $\mathrm{mrg}(t_0, \ldots, t_{k-1})$ is accepting. To establish that the Verifier-players have always used the Nonempty-case, we proceed by induction on the rounds. We show that, in Round r, Verifier i for each $i \in I_\exists$ has so far produced a trace prefix of length r such that it is possible to successfully continue the prefix in a way that allows the Verifier-players with higher indices to also continue successfully. Successful simply meaning the traces can be continued in a way such that $\mathcal{A}_{k-1}^{\mathfrak{T}}$ accepts. This implies that they can also use the Nonempty-case in the next round. ◀

Combining Lemma 8 and Lemma 11, we obtain our main result of this section for formulas φ where the maximal trace quantifier-free subformula ψ is a safety property.

▶ **Theorem 13.** *Let* Ξ, $\mathcal{P}$ *be as in Definition 10, and let* ψ *be a safety property. The coalition of Verifier-players has a winning collection of strategies for* $\mathcal{G}(\mathfrak{T}^{\mathcal{P}}, \varphi^{\mathcal{P},\Xi})$ *if and only if* $\mathfrak{T} \models \varphi$.

5 Complete Prophecies for ω-Regular Properties

In this section, we show how to construct complete prophecies for trace quantifier-free ψ beyond the safety fragment. In the safety case, prophecies indicate moves which prevent the Verifier-players from losing. In the case of safety, not losing equals winning. However for general ω-regular objectives, this is not enough. It is also necessary to (again and again) make progress towards satisfying the acceptance condition.

In this section, we assume that the automata $\mathcal{A}_i^{\mathfrak{T}}$ are *Rabin automata*. The class of languages recognized by (deterministic respectively nondeterministic) Rabin automata is the class of ω-regular languages. A (deterministic) Rabin automaton is a (deterministic)

[6] For $j < i$, she actually has access to $r+1$ letters, but our construction is independent of the last ones.

ω-automaton whose acceptance condition is given as a set $\{(B_1, F_1), \dots, (B_m, F_m)\}$ of pairs of sets B_i, F_i of states. A run is accepting if there exists an i such that the run visits states in B_i only finitely many times and states in F_i infinitely many times.

5.1 One-Pair Rabin Automata

First, we present our construction for one-pair Rabin automata, then show how to generalize to Rabin automata with an arbitrary number of pairs in Section 5.2. Here, we assume that for *each* $i \in I_\exists$, the automaton $\mathcal{A}_i^{\mathfrak{T}}$ is a one-pair Rabin automaton, not only $\mathcal{A}_k^{\mathfrak{T}}$. We always denote the single pair as (B, F), it will be clear from the context to which automaton it belongs. Intuitively, making progress towards winning in a one-pair Rabin automaton means avoiding states in B and visiting states in F.

Let κ be an infinite run of such an automaton. We define $\text{last}_B(\kappa)$ as $\bot$ if κ never visits B, as i if $\kappa(i)$ is the last visit of κ in B, and as ∞ if κ visits B infinitely many times (meaning there is no last visit). Furthermore, we define $\text{first}_F(\kappa)$ as i if $\kappa(i)$ is the first visit of κ in F and as ∞ if κ never visits F. Finally, we define the value $\text{val}(\kappa)$ of a run κ as $\text{first}_F(\kappa)$ if $\text{last}_B(\kappa) = \bot$ and as $\max\{\text{last}_B(\kappa), \text{first}_F(\kappa)\}$ otherwise. Given two runs κ, κ', we say that κ "makes progress faster" than κ' if $\text{val}(\kappa) < \text{val}(\kappa')$.

Before we can define our new prophecies, we need additional notation that gives us the value of a best run (over several traces) if we fix all but the last trace. For each $i \in I_\exists$, each vertex v of $\mathfrak{T}$, each state q of $\mathcal{A}_i^{\mathfrak{T}}$, and traces $t_0, t_1, \dots, t_{i-1} \in \text{Tr}(\mathfrak{T})$, we define

$$\text{opt}_i(q, v, t_0, \dots, t_{i-1}) = \min_{t_i \in \text{Tr}(\mathfrak{T}_{\text{S}(v)})} \text{val}(\kappa_i),$$

where κ_i is the run of $(\mathcal{A}_i^{\mathfrak{T}})_q$ on $\text{mrg}(t_0, \dots, t_{i-1}, t_i)$.

We are now ready to introduce our prophecies.

▶ **Definition 14** (Prophecy construction). *For each $i \in I_\forall$, vectors $\bar{u} = (v_0, v_1, \dots, v_{i+1})$ and $\bar{v} = (v'_0, v'_1, \dots, v'_{i+1})$ of vertices such that $v'_j \in \text{S}(v_j)$ for $j \le i+1$, and each vector $\bar{q} = (q_1, q_3, \dots, q_{i+1})$ of states q_j of $\mathcal{A}_j^{\mathfrak{T}}$, we define*

$$(\mathfrak{P}_i)_{\bar{q}}^{\bar{u},\bar{v}} = \{\text{mrg}(t_0, t_2, \dots, t_i) \mid t_0 \in \text{Tr}(\mathfrak{T}_{v'_0}), t_2 \in \text{Tr}(\mathfrak{T}_{v'_2}), \dots, t_i \in \text{Tr}(\mathfrak{T}_{v'_i}), \text{ and}$$
$$\text{there exist } t_1 \in \text{Tr}(\mathfrak{T}_{v'_1}), t_3 \in \text{Tr}(\mathfrak{T}_{v'_3}), \dots, t_{i+1} \in \text{Tr}(\mathfrak{T}_{v'_{i+1}}) \text{ such that}$$
$$\text{the run } \kappa_j \text{ of } (\mathcal{A}_j^{\mathfrak{T}})_{q_j} \text{ on } \text{mrg}(t_0, \dots, t_j) \text{ is accepting and satisfies}$$
$$\text{val}(\kappa_j) = \text{opt}_j(q_j, v_j, t_0, \dots, t_{j-1}) \text{ for all odd } j \le i+1\}.$$

Intuitively, the prophecy $(\mathfrak{P}_i)_{\bar{q}}^{\bar{u},\bar{v}}$ indicates that it is safe for Verifier $i+1$ to move from v_{i+1} to v'_{i+1} as before, and additionally, among all successors of v_{i+1}, picking v'_{i+1} allows Verifier $i+1$ to make the most progress. This prophecy is intended to be used, if for odd $j < i+1$, Verifier j already picked the move v_j to v'_j which was optimal. In particular, the prophecy repeats the prophecies made (in the same round) for Verifier j with odd $j < i+1$.

These prophecies are ω-regular.

▶ **Lemma 15.** *For each $i \in I_\forall$, each vector $\bar{q}$ of states and all vectors $\bar{u}, \bar{v}$ such that the ℓ-th vertex of $\bar{v}$ is a successor of the ℓ-th vertex of $\bar{u}$, the set $(\mathfrak{P}_i)_{\bar{q}}^{\bar{u},\bar{v}}$ is ω-regular.*

We set up the manipulated system and formula, then state the completeness result.

▶ **Definition 16.** *We define $\Xi = (\Xi_i)_{i \in I_\forall}$ and $\mathcal{P} = (\text{P}_i)_{i \in I_\forall}$. Let Ξ_i be the set of QPTL formulas that contains sentences $(\xi_i)_{\bar{q}}^{\bar{u},\bar{v}}$ for each vector $\bar{q}$ of states and vectors $\bar{u}, \bar{v}$ such that the ℓ-th vertex of $\bar{v}$ is a successor of the ℓ-th vertex of $\bar{u}$. The sentence $(\xi_i)_{\bar{q}}^{\bar{u},\bar{v}}$ expresses the prophecy $(\mathfrak{P}_i)_{\bar{q}}^{\bar{u},\bar{v}}$ using only even trace variables π_j with $j \le i$.*

Moreover, let P_i *be such that it contains the prophecy variable* $(\mathfrak{p}_i)_{\bar{q}}^{\bar{u},\bar{v}}$ *for each vector* $\bar{q}$ *of states and vectors* $\bar{u}, \bar{v}$ *such that the* ℓ*-th vertex of* $\bar{v}$ *is a successor of the* ℓ*-th vertex of* $\bar{u}$*. The prophecy variable* $(\mathfrak{p}_i)_{\bar{q}}^{\bar{u},\bar{v}}$ *corresponds to* $(\xi_i)_{\bar{q}}^{\bar{u},\bar{v}}$*.*

▶ **Lemma 17.** *Assume* $\mathfrak{T} \models \varphi$ *and each* $\mathcal{A}_i^{\mathfrak{T}}$ *is a one-pair Rabin automaton. Then the coalition of Verifier-players has a winning collection of strategies in* $\mathcal{G}(\mathfrak{T}^{\mathcal{P}}, \varphi^{\mathcal{P},\Xi})$ *with* Ξ *and* $\mathcal{P}$ *as in Definition 16.*

Our proof of this result generalizes the one for the safety case (see Lemma 11), using the same notation introduced there: We define a collection of strategies for the Verifier-players and then show, that $\mathfrak{T} \models \varphi$ implies that this collection is winning in $\mathcal{G}(\mathfrak{T}^{\mathcal{P}}, \varphi^{\mathcal{P},\Xi})$.

Assume the play is in Round (r, i) for $r \geq 0$ and $i \in I_\exists$. Thus, it is in a position

$$((v'_0, \ldots, v'_{i-1}, v_i, \ldots, v_{k-1}), q, i)$$

and Verifier i has to move. We review some information Verifier i can use to base her choice on. In Round r, Verifier i has access to the first r letters of the traces $\lambda_{\mathrm{AP}}(\rho_0), \lambda_{\mathrm{AP}}(\rho_1), \ldots, \lambda_{\mathrm{AP}}(\rho_i)$ induced by the play[6]. Let q_j be the state that $\mathcal{A}_j^{\mathfrak{T}}$ has reached on $\mathrm{mrg}(\lambda_{\mathrm{AP}}(\rho_0)[0,r), \lambda_{\mathrm{AP}}(\rho_1)[0,r), \ldots, \lambda_{\mathrm{AP}}(\rho_j)[0,r))$ for odd $j \leq i$. Also, let $((v_0, \ldots, v_{i-1}, v_i, \ldots, v_{k-1}), q, 0)$ be the position reached in Round $(r, 0)$.

▶ **Definition 18** (Strategy construction)**.** *Let*

$$M_i(\bar{u}, \bar{v}, \bar{q}) = \{v'_i \in \mathrm{S}(v_i) \mid (\mathfrak{p}_{i-1})_{\bar{q}}^{\bar{u},\bar{w}} \in \mathrm{Prophecies}(v'_{i-1})\}$$

where $\bar{u} = (v_0, \ldots, v_i)$*,* $\bar{v} = (v'_0, \ldots, v'_{i-1}, v_i)$*,* $\bar{w} = (v'_0, \ldots, v'_{i-1}, v'_i)$ *and* $\bar{q} = (q_1, q_3, \ldots, q_i)$*.*

If $M_i(\bar{u}, \bar{v}, \bar{q})$ *is nonempty, then Verifier* i *can move to any* v'_i *in the set (the "*Nonempty*-case"). If it is empty, then Verifier* i *can move to any* $v'_i \in \mathrm{S}(v_i)$ *(the "*Empty*-case").*

Now, Lemma 17 can be proven by generalizing the proof of Lemma 11. Then, we directly obtain our main result by combining Lemma 8 and Lemma 17.

▶ **Theorem 19.** *Let* Ξ *and* $\mathcal{P}$ *as in Definition 16, and assume that each* $\mathcal{A}_i^{\mathfrak{T}}$ *is a one-pair Rabin automaton. The coalition of Verifier-players has a winning collection of strategies for* $\mathcal{G}(\mathfrak{T}^{\mathcal{P}}, \varphi^{\mathcal{P},\Xi})$ *if and only if* $\mathfrak{T} \models \varphi$*.*

5.2 Beyond One-Pair Rabin Automata

We have proven Theorem 19 for the case that all automata referred to in the prophecies are one-pair Rabin automata. We sketch how to drop this constraint closely following Beutner and Finkbeiner [5].

A Rabin automaton $\mathcal{R}$ with pairs $(B_1, F_1), \ldots, (B_m, F_m)$ can be seen as a union of m one-pair Rabin automata (B_i, F_i)-$\mathcal{R}$. A run that is accepting in (B_i, F_i)-$\mathcal{R}$ is also accepting in $\mathcal{R}$ and each accepting run of $\mathcal{R}$ is accepting in some (B_i, F_i)-$\mathcal{R}$. Before we present how to go from one-pair to general Rabin automata, we recall that the prophecies exclusively talk about traces provided by Falsifier. Hence, if Falsifier signals that his future traces belong to prophecy $\mathcal{P}$ which, for example, implies that Rabin automaton $\mathcal{R}$ accepts them, then Falsifier can also specify the pair(s) for which $\mathcal{R}$ will accept. No further context from the Verifier-players is needed.

The idea is to annotate our prophecies (which refer to $\mathcal{A}_1^{\mathfrak{T}}, \mathcal{A}_3^{\mathfrak{T}}, \ldots, \mathcal{A}_{k-1}^{\mathfrak{T}}$) with the indices of the pairs used for acceptance. More concretely, for a prophecy $(\mathfrak{P}_i)_{\bar{q}}^{\bar{u},\bar{v}}$, we introduce prophecies of the form $(\mathfrak{P}_i)_{\bar{q},\bar{a}}^{\bar{u},\bar{v}}$, where $\bar{a}$ is a vector of indices of the same length as $\bar{q}$. Say the

j-th entry of $\bar{q}$ is q_ℓ, then the prophecy $(\mathfrak{P}_i)_{\bar{q}}^{\bar{u},\bar{v}}$ refers to $(\mathcal{A}_\ell^{\mathfrak{T}})_{q_\ell}$ (among other automata), and say the j-th entry of $\bar{a}$ is a_j, then $(\mathfrak{P}_i)_{\bar{q},\bar{a}}^{\bar{u},\bar{v}}$ refers to the one-pair Rabin automaton (B_{a_j}, F_{a_j})-$(\mathcal{A}_\ell^{\mathfrak{T}})_{q_\ell}$ (among other automata). Except for the refined prophecies, no other changes to the game are required.

In a play of the game $\mathcal{G}(\mathfrak{T}^{\mathcal{P}}, \varphi^{\mathcal{P},\Xi})$, Falsifier specifies with his prophecies for which pairs the Rabin automata $\mathcal{A}_1^{\mathfrak{T}}, \mathcal{A}_3^{\mathfrak{T}}, \ldots, \mathcal{A}_{k-1}^{\mathfrak{T}}$ can accept.

In Round $(0,0)$, it is specified for $\mathcal{A}_1^{\mathfrak{T}}$, in Round $(0,1)$, Verifier 1 chooses one option specified by Falsifier for $\mathcal{A}_1^{\mathfrak{T}}$. That choice is simply made by her move. In Round $(0,0)$, Falsifier moves from $\bullet$ to v_0, and in Round $(0,1)$, Verifier 1 moves from $\bullet$ to v_1. Say Prophecies(v_0) indicates that (B_i, F_i)-$\mathcal{A}_1^{\mathfrak{T}}$ accepts when Verifier 1 moves to v_1, then she has chosen (B_i, F_i)-$\mathcal{A}_1^{\mathfrak{T}}$. She then (in order be guaranteed to build a winning play) has to stick to it for the rest of the play, meaning that from now on all her moves must have a target vertex v such that the prophecy given by Falsifier in the move before indicates that (B_i, F_i)-$\mathcal{A}_1^{\mathfrak{T}}$ accepts if she moves to v.

In Round $(0,2)$, it is specified also for $\mathcal{A}_3^{\mathfrak{T}}$, in Round $(0,3)$, Verifier 3 chooses one option for $\mathcal{A}_3^{\mathfrak{T}}$ and then has to stick to it for the rest of the play. Additionally, she must pick the option chosen by Verifier 1 for $\mathcal{A}_1^{\mathfrak{T}}$ to ensure consistency. In Round $(0,2)$, Falsifier moves from $\bullet$ to v_2. For example, say Verifier 3 has the option to move from $\bullet$ to v', v'', or v''' with Prophecies(v_2) indicating that (B_i, F_i)-$\mathcal{A}_1^{\mathfrak{T}}$ and (B_j, F_j)-$\mathcal{A}_3^{\mathfrak{T}}$ accept if she moves to v', that (B_k, F_k)-$\mathcal{A}_1^{\mathfrak{T}}$ and (B_j, F_j)-$\mathcal{A}_3^{\mathfrak{T}}$ accept if she moves to v'', and that (B_i, F_i)-$\mathcal{A}_1^{\mathfrak{T}}$ and (B_ℓ, F_ℓ)-$\mathcal{A}_3^{\mathfrak{T}}$ accept if she moves to v'''. Then she must pick v' or v''' because Verifier 1 has committed to (B_i, F_i)-$\mathcal{A}_1^{\mathfrak{T}}$ which Verifier 3 must honor. If Verifier 3 moves to v' she picks (B_j, F_j)-$\mathcal{A}_3^{\mathfrak{T}}$, if she moves to v''' she picks (B_ℓ, F_ℓ)-$\mathcal{A}_3^{\mathfrak{T}}$.

The same principle applies for the next rounds. After Round $(0, k-1)$, for each automaton a pair has been fixed to which the Verifier-players stick for the rest of the play.

We end by substantiating the claim in Footnote 4 on Page 8 that our construction can also be applied to formulas with arbitrary, i.e., not strictly alternating, trace quantifier prefixes.

▶ Remark 20. In case of of arbitrary trace quantifier prefixes, we have one player in the coalition for each maximal block of existentially quantified trace variables in the prefix, who selects traces for these variables. For example, for a formula of the form

$$\forall\pi_0\exists\pi_1\exists\pi_2\forall\pi_3\exists\pi_4\exists\pi_5.\ \psi$$

with trace quantifier-free ψ, we have just two players (say, Player 12 and Player 45) in the coalition. Player 12 is in charge of the variables π_1 and π_2, and Player 45 is in charge of the variables π_4 and π_5. These two players are in a coalition against Falsifier, who is in charge of the variables π_0 and π_3. In each round, Falsifier picks a move for π_0, then Player 12 picks a move for π_1 and a move for π_2, then Falsifier picks a move for π_3, and then Player 45 picks a move for π_4 and a move for π_5.

Note that just adding universally quantified dummy trace variables between any pair of consecutive existentially quantified trace variables is logically equivalent, but increases the complexity of our approach (as that depends on the number of players).

6 Conclusion

We have presented the first sound and complete game-based algorithm for full HyperLTL (even HyperQPTL) model-checking via multi-player games with hierarchical information, thereby extending the prophecy-based framework of Coenen et al. and Beutner and Finkbeiner from the $\forall^*\exists^*$-fragment to arbitrary quantifier prefixes. Similarly, we have extended Winter and

Zimmermann's game-based characterization of the existence of computable Skolem functions witnessing $\mathfrak{T} \models \varphi$. One way of understanding our result is that we compute finite-state implementations of Skolem functions via transducers with ω-regular lookahead.

In further work, we aim to exhibit lower bounds on the number of prophecies needed for completeness. Recall that HyperLTL model-checking is TOWER-complete. Even an elementary number of prophecies, while unlikely, would not contradict any complexity-lower bounds, as solving multi-player games of hierarchical information is already TOWER-complete (in the number of players). However, as it is, the number of prophecies we use is only bounded by an n-fold exponential, where n is linear in the number of quantifier alternations (taking both trace and propositional quantification into account), as the size of the automata $\mathcal{A}_i^{\mathfrak{T}}$ grows like that.

One of the motivations for the introduction of the game-based approach using prophecies is the need for complementation of ω-automata in the classical automata-based HyperLTL model-checking algorithm. For $\forall^*\exists^*$-formulas, the game-based approach "only" requires deterministic ω-automata, which can be directly computed from the quantifier-free part, which is essentially an LTL formula. On the other hand, the automata-based approach needs automata complementation, already for $\forall^*\exists^*$-formulas. For arbitrary quantifier alternations, both approaches require complementation. In future work, we aim to study whether nondeterministic or history-deterministic automata are sufficient to construct complete prophecies.

Finally, we are currently extending the techniques developed here to even more expressive logics, e.g., HyperRecHML, recursive Hennessy-Milner logic with trace quantification [2].

References

1 Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82(2):253–284, 1991. doi:10.1016/0304-3975(91)90224-P.

2 Luca Aceto, Antonis Achilleos, Elli Anastasiadi, and Adrian Francalanza. Monitoring hyperproperties with circuits. In Mohammad Reza Mousavi and Anna Philippou, editors, *FORTE 2022*, volume 13273 of *LNCS*, pages 1–10. Springer, 2022. doi:10.1007/978-3-031-08679-3_1.

3 Rajeev Alur, Emmanuel Filiot, and Ashutosh Trivedi. Regular transformations of infinite strings. In *LICS 2012*, pages 65–74. IEEE Computer Society, 2012. doi:10.1109/LICS.2012.18.

4 Dietmar Berwanger, Anup Basil Mathew, and Marie van den Bogaard. Hierarchical information and the synthesis of distributed strategies. *Acta Informatica*, 55(8):669–701, 2018. doi:10.1007/S00236-017-0306-5.

5 Raven Beutner and Bernd Finkbeiner. Prophecy variables for hyperproperty verification. In *CSF 2022*, pages 471–485. IEEE, 2022. doi:10.1109/CSF54842.2022.9919658.

6 Michael R. Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K. Micinski, Markus N. Rabe, and César Sánchez. Temporal logics for hyperproperties. In Martín Abadi and Steve Kremer, editors, *POST 2014*, volume 8414 of *LNCS*, pages 265–284. Springer, 2014. doi:10.1007/978-3-642-54792-8_15.

7 Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *J. Comput. Secur.*, 18(6):1157–1210, 2010. doi:10.3233/JCS-2009-0393.

8 Norine Coenen, Bernd Finkbeiner, César Sánchez, and Leander Tentrup. Verifying hyperliveness. In Isil Dillig and Serdar Tasiran, editors, *CAV 2019, Part I*, volume 11561 of *LNCS*, pages 121–139. Springer, 2019. doi:10.1007/978-3-030-25540-4_7.

9 Samuel Eilenberg. *Automata, languages, and machines. A*. Pure and applied mathematics. Academic Press, 1974. URL: https://www.worldcat.org/oclc/310535248.

10 E. Allen Emerson and Joseph Y. Halpern. "Sometimes" and "not never" revisited: on branching versus linear time temporal logic. *J. ACM*, 33(1):151–178, 1986. doi:10.1145/4904.4999.

11 Joost Engelfriet. Top-down tree transducers with regular look-ahead. *Math. Syst. Theory*, 10:289–303, 1977. doi:10.1007/BF01683280.

12 Emmanuel Filiot and Sarah Winter. Synthesizing computable functions from rational specifications over infinite words. *Int. J. Found. Comput. Sci.*, 35(01n02):179–214, 2024. doi:10.1142/S012905412348009X.

13 Bernd Finkbeiner, Christopher Hahn, Jana Hofmann, and Leander Tentrup. Realizing omega-regular hyperproperties. In Shuvendu K. Lahiri and Chao Wang, editors, *CAV 2020, Part II*, volume 12225 of *LNCS*, pages 40–63. Springer, 2020. doi:10.1007/978-3-030-53291-8_4.

14 Bernd Finkbeiner, Markus N. Rabe, and César Sánchez. Algorithms for Model Checking HyperLTL and HyperCTL*. In Daniel Kroening and Corina S. Pasareanu, editors, *CAV 2015, Part I*, volume 9206 of *LNCS*, pages 30–48. Springer, 2015. doi:10.1007/978-3-319-21690-4_3.

15 Erich Grädel, Wolfgang Thomas, and Thomas Wilke, editors. *Automata, Logics, and Infinite Games: A Guide to Current Research*, volume 2500 of *LNCS*. Springer, 2002. doi:10.1007/3-540-36387-4.

16 Frederick A. Hosch and Lawrence H. Landweber. Finite delay solutions for sequential conditions. In *ICALP 1972*, pages 45–60, 1972.

17 Felix Klein and Martin Zimmermann. How much lookahead is needed to win infinite games? *Log. Methods Comput. Sci.*, 12(3), 2016. doi:10.2168/LMCS-12(3:4)2016.

18 Corto Mascle and Martin Zimmermann. The keys to decidable HyperLTL satisfiability: Small models or very simple formulas. In Maribel Fernández and Anca Muscholl, editors, *CSL 2020*, volume 152 of *LIPIcs*, pages 29:1–29:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.CSL.2020.29.

19 Daryl McCullough. Noninterference and the composability of security properties. In *SSP 1988*, pages 177–186. IEEE Computer Society, 1988. doi:10.1109/SECPRI.1988.8110.

20 Amir Pnueli. The temporal logic of programs. In *FOCS 1977*, pages 46–57. IEEE, October 1977. doi:10.1109/SFCS.1977.32.

21 Amir Pnueli and Roni Rosner. Distributed reactive systems are hard to synthesize. In *FOCS 1990, Volume II*, pages 746–757. IEEE Computer Society, 1990. doi:10.1109/FSCS.1990.89597.

22 Markus N. Rabe. *A temporal logic approach to information-flow control*. PhD thesis, Saarland University, 2016. URL: http://scidok.sulb.uni-saarland.de/volltexte/2016/6387/.

23 A. Prasad Sistla, Moshe Y. Vardi, and Pierre Wolper. The complementation problem for Büchi automata with appplications to temporal logic. *Theor. Comput. Sci.*, 49:217–237, 1987. doi:10.1016/0304-3975(87)90008-9.

24 Sarah Winter and Martin Zimmermann. Finite-state strategies in delay games. *Inf. Comput.*, 272:104500, 2020. doi:10.1016/J.IC.2019.104500.

25 Sarah Winter and Martin Zimmermann. Prophecies all the way: Game-based model-checking for HyperQPTL beyond ∀*∃*. *arXiv*, 2504.08575, 2025. doi:10.48550/arXiv.2504.08575.

Reachability in Vector Addition System with States Parameterized by Geometric Dimension

Yangluo Zheng
BASICS, Shanghai Jiao Tong University, China

Abstract

The geometric dimension of a vector addition system with states (VASS), emerged in Leroux and Schmitz (2019) and formalized by Fu, Yang, and Zheng (2024), quantifies the dimension of the vector space spanned by cycle effects in the system. This paper examines the VASS reachability problem through the lens of geometric dimension, revealing key differences from the traditional dimensional parameterization. Notably, we establish that the reachability problem for both geometrically 1-dimensional and 2-dimensional VASS is PSPACE-complete, achieved by extending the pumping technique initially proposed by Czerwiński et al. (2019).

2012 ACM Subject Classification Theory of computation → Models of computation; Theory of computation → Logic and verification

Keywords and phrases Petri net, vector addition system, reachability, geometric dimension, pumping

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2025.38

Related Version *Full Version*: https://arxiv.org/abs/2412.14608

Funding The support from the National Science Foundation of China (62072299), Science and Technology Commission of Shanghai Municipality (24BC3200500) are acknowledged.

Acknowledgements The author would like to thank Wojciech Czerwiński, Łukasz Orlikowski and Qizhe Yang for a fruitful discussion which motivates Section 5. The author also thanks the members of BASICS for their interest. The efforts of anonymous reviewers are appreciated.

1 Introduction

Vector addition systems with states (VASSes), equivalent to Petri nets, serve as a fundamental model for concurrency [10]. A VASS extends finite automata with integer counters that cannot be zero-tested but must be kept non-negative. Central to the algorithmic theory of VASS is the *reachability problem*: determining if a run exists from one configuration to another. Due to its generic nature, numerous practical problems can be modeled via the reachability problem [19]. After decades of study, the computational complexity of the VASS reachability problem was settled to be Ackermann-complete [15, 7, 14]. However, when fixing the *dimension* – the number of counters, a gap remains in complexity bounds. For d-dimensional VASS where $d > 2$, reachability lies in F_d [12], the dth level of the Grzegorczyk hierarchy of complexity classes [17] (for $d = 3$ the upper bound has been improved to 2-EXPSPACE recently [8]), while F_d-hardness is achieved with $(2d + 3)$-dimensional VASS [5]. This gap is known to be closed only in low dimensions. PSPACE-completeness holds for 2-dimensional VASS under binary encoding [3, 6], and NL-completeness under unary encoding [9]. NP-completeness holds for 1-dimensional VASS under binary encoding [13], and again NL-completeness under unary encoding [9]. Dimension has traditionally served as the standard parameterization for reachability.

On the other hand, the structure of *cycles* was identified as a pivotal factor controlling the complexity of the VASS reachability problem [15]. The notion of *geometric dimension*, formalized in [12], measures the dimension of the *cycle space* – vector space spanned by all cycle effects. Insights from [15, 12] suggest that the F_d upper bound of the famous KMLST

© Yangluo Zheng;
licensed under Creative Commons License CC-BY 4.0

algorithm applies to geometric dimension d, not only to dimension d. In this work we propose geometric dimension as an alternative parameterization for the VASS reachability problem. In theory the geometric dimension is closer to the nature of the reachability problem, as suggested by [15] and [12], and the cycles provides a rich structure for analysis. Recently a work on 3-dimensional VASS also made use of the geometric dimension [8]. Moreover, the structure of cycles played an important role in a study on continuous VASSes [1] where the authors extended the technique of linear path schemes to study the reachability problem. In practice, as the system parameters might not necessarily be independent of each other, fixing geometric dimension rather than dimension allows one to introduce certain types of interconnections in system parameters for free, which could possibly make the model more expressive. For example, a bounded counter can be simulated by a pair of complementary counters such that transitions increasing one of them must simultaneously decrease the other by the same amount. One easily verifies that these two counters contribute at most 1 to the geometric dimension.

Our contribution

In this paper, we study the reachability problem in VASSes with fixed geometric dimensions. As mentioned before, the F_d upper bound in [12] directly applies to geometric dimension $d \geq 3$. Thus, the primary focus is on VASSes with geometric dimension ≤ 2. Our main contributions are the following theorems.

▶ **Theorem 1.1.** *Reachability in VASS of geometric dimension 2 is* PSPACE*-complete under binary encoding.*

Previous work [12] showing the semi-linearity of VASS reachability set in geometric dimension 2 utilized the technique of linear path schemes, which yielded merely an EXPSPACE upper bound (see [18] for an explicit statement). This is largely due to an exponential blow-up in the number of control states introduced by a dimension-reduction argument (see, e.g. [12, Lemma A.18]). In contrast, our proof relies on a pumping technique for 2-dimensional VASSes [6]. To apply this technique, we make use of the *sign-reflecting projection* proposed in [12], with some further properties developed in Subsection 4.1. Another tool called the *support projection* is introduced in Subsection 4.2. Combining these projection tools we establish a suitable coordinate system (Lemma 4.6) within the 2-dimensinoal cycle space of the VASS. This enables us to apply the arguments in [6] to obtain the PSPACE upper bound. Together with the PSPACE-hardness inherited from 2-dimensinoal VASS [3, 11], we conclude PSPACE-completeness. We mention that the projection tools in this paper do not provide a straightforward reduction from d-dimensional VASS of geometric dimension 2 to 2-dimensinoal VASS. But we will show such a reduction exists for $d = 3$ in Section 5.

Geometric dimensions lower than 2 are also studied in this paper:

▶ **Theorem 1.2.** *Reachability in VASS of geometric dimension 1 is* PSPACE*-complete, and that of geometric dimension 0 is* NP*-complete under binary encoding.*

Results in VASS of geometric dimension 1 and 0 are obtained by a re-examination of known results for VASS of dimension 1 and 2. These results show an interesting distinction in complexity of VASS reachability parameterized by dimension and by geometric dimension, as compared in Table 1.

In addition to the above complexity results, we also give an efficient (polynomial time) algorithm computing the geometric dimension of a VASS in Subsection 3.1.

Table 1 Complexity of VASS reachability parameterized by dimension and by geometric dimension.

	dimension d	geometric dimension d
$d = 0$	NL-complete (folklore)	**NP-complete**
$d = 1$	NP-complete [13]	**PSPACE-complete**
$d = 2$	PSPACE-complete [3]	**PSPACE-complete**
$d = 3$	PSPACE-hard [3], in 2-EXPSPACE[8]	PSPACE-hard [3], in TOWER $= \mathsf{F}_3$ [12]
$d \geq 4$	no known distinctions	

Organization

Section 2 fixes notations and definitions. Section 3 introduces the geometric dimension of a VASS and discusses some useful properties. Section 4 proves the PSPACE-completeness of geometrically 2-dimensional VASS. Section 5 gives a straightforward reduction from geometrically 2-dimensional 3-VASS to 2-VASS. Section 6 considers geometric dimensions lower than 2. Section 7 concludes the paper. Omitted proofs can be found in the full version.

2 Preliminaries

We use $\mathbb{N}, \mathbb{Z}, \mathbb{Q}$ to denote the set of natural numbers (non-negative integers), integers, and rational numbers respectively. Let $m \leq n$ be integers, we use $[m, n]$ to denote the set $\{m, m+1, \ldots, n\}$. And we abbreviate $[n]$ for $[1, n]$. For a d-dimensional vector $\boldsymbol{v} \in \mathbb{Q}^d$, we write $\boldsymbol{v}(i)$ for its ith component, and we use its maximum norm $\|\boldsymbol{v}\| := \max_{i \in [d]} |\boldsymbol{v}(i)|$. The order $\leq$ is extended component-wise to vectors: we write $\boldsymbol{u} \leq \boldsymbol{v}$ if $\boldsymbol{u}(i) \leq \boldsymbol{v}(i)$ for all $i \in [d]$. Similarly we define the component-wise strict order $<$ for vectors. We write $\langle \boldsymbol{u}, \boldsymbol{v} \rangle = \sum_{i \in [d]} \boldsymbol{u}(i)\boldsymbol{v}(i)$ for their inner product. The *support* of a vector $\boldsymbol{v}$ is $\mathrm{supp}(\boldsymbol{v}) := \{i \in [d], \boldsymbol{v}(i) \neq 0\}$. The support of a set S of vectors is $\mathrm{supp}(S) := \bigcup_{\boldsymbol{v} \in S} \mathrm{supp}(\boldsymbol{v})$. A vector $\boldsymbol{v}$ is *positive* if $\boldsymbol{v} > \boldsymbol{0}$, it is *semi-positive* if $\boldsymbol{v} \geq \boldsymbol{0}$ and $\boldsymbol{v} \neq \boldsymbol{0}$. For a string $s = a_1 a_2 \ldots a_n \in \Sigma^*$ over an alphabet Σ, we write $s[i..j]$ for the substring $a_i a_{i+1} \ldots a_j$ of s.

2.1 Vector Addition System with States

Let $d \geq 0$ be an integer. A *d-dimensional vector addition system with states (d-VASS)* is a pair $G = (Q, T)$ where Q is a finite set of *states* and $T \subseteq Q \times \mathbb{Z}^d \times Q$ is a finite set of *transitions*. Clearly a VASS can also be viewed as a directed graph with edges labelled by integer vectors. Given a word $\pi = (p_1, \boldsymbol{a}_1, q_1)(p_2, \boldsymbol{a}_2, q_2) \ldots (p_n, \boldsymbol{a}_n, q_n) \in T^*$ over transitions, we say that π is a *path from p_1 to q_n* if $q_i = p_{i+1}$ for all $i = 1, \ldots, n-1$. It is a *cycle* if we further have $p_1 = q_n$. Such a path is usually presented in the following form:

$$\pi = p_1 \xrightarrow{t_1} q_1 \xrightarrow{t_2} q_2 \xrightarrow{t_3} \cdots \xrightarrow{t_n} q_n \tag{1}$$

where $t_i = (p_i, \boldsymbol{a}_i, q_i)$. The *effect* of π is defined to be $\Delta(\pi) := \sum_{i=1}^{n} \boldsymbol{a}_i$.

Size, traversal number and characteristic

The norm of a transition $t = (p, \boldsymbol{a}, q)$ is defined by $\|t\| := \|\boldsymbol{a}\|$. For a d-VASS $G = (Q, T)$ we write $\|T\| := \max\{\|t\| : t \in T\}$. We shall mainly consider VASS under binary encoding, so the *size* of G is given by $|G| := |Q| + d \cdot |T| \cdot \lceil \log(\|T\| + 1) \rceil + 1$.

We define the *traversal number* of G to be the maximal number of distinct states that can be visited (traversed) by a path in G, denoted by $\varsigma(G)$. We remark that $\varsigma(G)$ is an upper bound of (i) the length of any simple path/cycle, and (ii) the number of connected components visited by any path. Note also the trivial fact $\varsigma(G) \leq |Q|$.

The *characteristic* of G, denoted by $\chi(G)$, is defined to be $\chi(G) := \varsigma(G) \cdot \|T\|$. So $\chi(G)$ upper bounds the norm of effect of any simple path/cycle in G.

Semantics of VASSes

Let $G = (Q, T)$ be a d-VASS. A *configuration* c of G is a pair of a state $p \in Q$ and a vector $\boldsymbol{v} \in \mathbb{N}^d$, written as $c = p(\boldsymbol{v})$. We will often confuse a configuration with its vector. So we shall write, for example, $\|c\|$ for $\|\boldsymbol{v}\|$, and $c(i)$ for $\boldsymbol{v}(i)$. The semantics of G is defined as follows. For each transition $t = (p, \boldsymbol{a}, q) \in T$, the one-step transition relation $\xrightarrow{t}$ relates all pairs of configurations of the form $(p(\boldsymbol{u}), q(\boldsymbol{v}))$ where $\boldsymbol{u}, \boldsymbol{v} \in \mathbb{N}^d$ and $\boldsymbol{v} = \boldsymbol{u} + \boldsymbol{a}$. Then for a word $\pi = t_1 t_2 \dots t_n \in T^*$, the relation $\xrightarrow{\pi}$ is the composition $\xrightarrow{\pi} := \xrightarrow{t_1} \circ \cdots \circ \xrightarrow{t_n}$. So $p(\boldsymbol{u}) \xrightarrow{\pi} q(\boldsymbol{v})$ if and only if there are configurations $p_0(\boldsymbol{u}_0), \dots, p_n(\boldsymbol{u}_n) \in Q \times \mathbb{N}^d$ such that

$$p(\boldsymbol{u}) = p_0(\boldsymbol{u}_0) \xrightarrow{t_1} p_1(\boldsymbol{u}_1) \xrightarrow{t_2} \cdots \xrightarrow{t_n} p_n(\boldsymbol{u}_n) = q(\boldsymbol{v}). \tag{2}$$

Also, when $\pi = \epsilon$ is the empty word, the relation $\xrightarrow{\epsilon}$ is the identity relation over $Q \times \mathbb{N}^d$. Note that $\xrightarrow{\pi}$ is non-empty only if π is a path. When $p(\boldsymbol{u}) \xrightarrow{\pi} q(\boldsymbol{v})$ we also say that π induces (or is) a *run* from $p(\boldsymbol{u})$ to $q(\boldsymbol{v})$, and we write $\mathrm{src}(\pi) := p(\boldsymbol{u})$ and $\mathrm{trg}(\pi) := q(\boldsymbol{v})$ for the *source* and *target* of π. We emphasize that all configurations on this run lie in $\mathbb{N}^d$, and that they are uniquely determined by $p(\boldsymbol{u})$ and π. Finally, the *reachability relation* of G is defined to be $\xrightarrow{*} := \bigcup_{\pi \in T^*} \xrightarrow{\pi}$.

Reachability problem

The general reachability problem in VASS is formulated as follows:

Reachability in Vector Addition System with States
Input: A VASS $G = (Q, T)$, two configurations $p(\boldsymbol{u}), q(\boldsymbol{v})$ of G.
Question: Does $p(\boldsymbol{u}) \xrightarrow{*} q(\boldsymbol{v})$ hold in G?

It is a folklore that this problem can be reduced to the following one in polynomial time without affecting the dimension or the geometric dimension:

0-Reachability in Vector Addition System with States
Input: A $G = (Q, T)$, two states $p, q \in Q$.
Question: Does $p(\mathbf{0}) \xrightarrow{*} q(\mathbf{0})$ hold in G?

Thus, in this paper we shall mainly care about runs starting from $\mathbf{0}$ and ending at $\mathbf{0}$. Such a run is called a $\mathbf{0}$-*run* in the following.

Reverse of VASS

For a VASS $G = (Q, T)$ we define its *reverse* as the VASS $G^{\mathrm{rev}} = (Q, T^{\mathrm{rev}})$ where $T^{\mathrm{rev}} := \{(q, -\boldsymbol{a}, p) : (p, \boldsymbol{a}, q) \in T\}$. The *reverse* $\mathrm{rev}(\pi)$ of a path (or a run) π is defined naturally by reversing the order of transitions in π, switching the source and target states of those transitions, and negating their effects. We note that $p(\boldsymbol{u}) \xrightarrow{\pi} q(\boldsymbol{v})$ in G if and only if $q(\boldsymbol{v}) \xrightarrow{\mathrm{rev}(\pi)} p(\boldsymbol{u})$ in G^{rev}.

3 Geometric Dimension

▶ **Definition 3.1.** *Let G be a d-VASS. The* cycle space *of G is the vector space* $\mathrm{Cyc}(G) \subseteq \mathbb{Q}^d$ *spanned by the effects of all cycles in G, that is:* $\mathrm{Cyc}(G) := \mathrm{span}\{\Delta(\theta) : \theta \text{ is a cycle in } G\}$.

The dimension of the cycle space of G is called the geometric dimension *of G, denoted by* $\mathrm{gdim}(G) := \dim(\mathrm{Cyc}(G))$. *We say G is* geometrically k-dimensional *if* $\mathrm{gdim}(G) \leq k$.

It should be noticed that the cycle space of a VASS is indeed spanned by the effects of all simple cycles in it.

▶ **Lemma 3.2.** *Let G be a d-VASS, then* $\mathrm{Cyc}(G)$ *equals to the vector space spanned by the effects of all simple cycles in G, that is,* $\mathrm{Cyc}(G) = \mathrm{span}\{\Delta(\beta) : \beta \text{ is a simple cycle in } G\}$.

Proof. Just note that the effect of every cycle is a finite sum of effects of simple cycles. ◀

A naive algorithm that computes the geometric dimension of a VASS by enumerating all simple cycles in it requires PSPACE. Indeed, we show here a more efficient algorithm which computes $\mathrm{gdim}(G)$ in polynomial time.

3.1 Computing Geometric Dimension

We present a stronger algorithm that given a VASS G as input, computes a basis for $\mathrm{Cyc}(G)$. Observe that every cycle lies within some maximal strongly connected component (SCC) of G. Once we have an algorithm that computes a basis for the cycle spaces of every SCC of G, a basis for $\mathrm{Cyc}(G)$ is just a maximal linearly independent subset of the union of these bases, which can be computed by Gaussian elimination in polynomial time. So the problem reduces to computing a basis for $\mathrm{Cyc}(G)$ in case G is strongly connected.

Fix a strongly connected d-VASS $G = (Q, T)$. We introduce an operation called *cycle shrinking*. Let θ be a simple cycle in G that is not a self-loop and has the form

$$\theta = p_0 \xrightarrow{t_1} p_1 \xrightarrow{t_2} \cdots \xrightarrow{t_n} p_n = p_0. \tag{3}$$

First we define a "shift function" $s : Q \to \mathbb{Z}^d$ as follows. If $q = p_k$ for some $k \in [n]$, we set $s(q) := \Delta(\theta[1..k]) = \Delta(t_1) + \ldots + \Delta(t_k)$; otherwise we set $s(q) := \mathbf{0}$. Note that $s(p_0) = s(p_n) = \Delta(\theta)$. Let $P = \{p_1, \ldots, p_n\}$ be the set of states that occurs on θ. Now we define a new VASS $G/\theta = (Q^\theta, T^\theta)$ that "shrinks" θ into a single states as follows.

- $Q^\theta := (Q \setminus P) \cup \{\theta\}$.
- Let $h : Q \to Q^\theta$ be defined as $h(q) = \theta$ if $q \in P$ and $h(q) = q$ otherwise. The transitions are given by $T^\theta := \{(h(p), s(p) + \boldsymbol{a} - s(q), h(q)) : (p, \boldsymbol{a}, q) \in T\}$.

It should be clear that $|Q^\theta| < |Q|$ as θ is not a self-loop, and that G/θ can be constructed in polynomial time given G and θ. Note that $\|T^\theta\| \leq (2|Q| + 1)\,\|T\|$, so the size of G/θ is bounded by $|G/\theta| \leq |G| + d \cdot |T| \cdot \lceil \log(2|Q| + 1) \rceil$. Observe that h is a graph homomorphism from G onto G/θ. Thus G/θ is strongly connected as long as G is. Besides, we can show that G/θ preserves the cycle space of G.

▶ **Proposition 3.3.** $\mathrm{Cyc}(G/\theta) = \mathrm{Cyc}(G)$.

Now the algorithm for computing a basis of $\mathrm{Cyc}(G)$ where G is strongly connected should be clear. As listed in Algorithm 1, we repeatedly shrink a cycle in G until there remains only one state. Then its cycle space is the span of effects of self-loops in it. A basis of $\mathrm{Cyc}(G)$ can be computed using Gaussian elimination. Since a cycle shrinking reduces the number of states by at least one, after at most $|Q|$ iterations we must stop with a single state remained. Note that the size of VASS is always bounded by $|G| + d \cdot |Q| \cdot |T| \lceil \log(2|Q| + 1) \rceil \leq |G|^2$ in each iteration. Thus the algorithm runs in polynomial time.

Algorithm 1 CycleSpaceBasis.

input : a VASS G which is strongly connected
output : basis of $\mathrm{Cyc}(G)$

while *G contains more than one state* **do**
  $\theta \leftarrow$ a simple cycle in G that is not a self-loop
  $G \leftarrow G/\theta$
end
$U \leftarrow$ the set of effects of all self-loops in G
return *a basis of U found by Gaussian elimination*

3.2 Geometry of Reachability Sets and Runs

Given a VASS $G = (Q, T)$ and a configuration $p(\boldsymbol{u}) \in Q \times \mathbb{N}^d$, we write $\mathrm{Reach}_G(p(\boldsymbol{u}))$ for all configurations that is reachable from $p(\boldsymbol{u})$: $\mathrm{Reach}_G(p(\boldsymbol{u})) := \{q(\boldsymbol{v}) \in Q \times \mathbb{N}^d : p(\boldsymbol{u}) \xrightarrow{*} q(\boldsymbol{v})\}$. The next lemma shows that the "dimension" of any reachable set is bounded by $\mathrm{gdim}(G)$, in the sense that it is contained in a finite union of affine copies of $\mathrm{Cyc}(G)$. Here the sum of a vector $\boldsymbol{v} \in \mathbb{Q}^d$ and a set $S \subseteq \mathbb{Q}^d$ is defined as $\boldsymbol{v} + S := \{\boldsymbol{v} + \boldsymbol{s} : \boldsymbol{s} \in S\}$.

▶ **Lemma 3.4.** *Let $G = (Q, T)$ be a d-VASS, $p(\boldsymbol{u}) \in Q \times \mathbb{N}^d$ be a configuration of G. Then*

$$\mathrm{Reach}_G(p(\boldsymbol{u})) \subseteq Q \times \bigcup_{\substack{\boldsymbol{z} \in \mathbb{Z}^d \\ \|\boldsymbol{z}\| \leq \chi(G)}} \boldsymbol{u} + \mathrm{Cyc}(G) + \boldsymbol{z}. \tag{4}$$

In other words, for any configuration $q(\boldsymbol{v}) \in Q \times \mathbb{N}^d$ with $p(\boldsymbol{u}) \xrightarrow{*} q(\boldsymbol{v})$, we have $\boldsymbol{v} = \boldsymbol{u} + \boldsymbol{c} + \boldsymbol{z}$ for some $\boldsymbol{c} \in \mathrm{Cyc}(G)$ and $\boldsymbol{z} \in \mathbb{Z}^d$, where $\|\boldsymbol{z}\| \leq \chi(G)$.

Note that one may need exponentially many (roughly $O(\chi(G)^d)$) affine copies of $\mathrm{Cyc}(G)$ to cover $\mathrm{Reach}_G(p(\boldsymbol{u}))$. The next lemma shows that any fixed run from $p(\boldsymbol{u})$ is confined in, however, at most $|Q|$ affine copies of $\mathrm{Cyc}(G)$.

▶ **Lemma 3.5.** *Let $G = (Q, T)$ be a d-VASS. For any run π in G with source $p(\boldsymbol{u}) \in Q \times \mathbb{N}^d$, there is a function $f_\pi : Q \to \mathbb{Z}^d$ such that for every configuration $q(\boldsymbol{v})$ occurring on π, we have $\boldsymbol{v} \in \boldsymbol{u} + \mathrm{Cyc}(G) + f_\pi(q)$. Moreover, $\|f_\pi(q)\| \leq \chi(G)$ for every $q \in Q$.*

These two lemmas follow easily from the fact that we can view a run as a simple path interleaved with cycles. Their proofs can be found in the full version of this paper.

4 Geometrically 2-Dimensional VASS

In this section we focus exclusively on geometrically 2-dimensional VASSes. We prove that reachability in geometrically 2-dimensional VASSes is PSPACE-complete. The lower bound is a simple corollary of [3, Lemma 20], so most effort will be devoted to the upper bound. Our proof is based on the pumping technique proposed in [6] for 2-VASSes, where they showed that every run in a 2-VASS is either *thin* – confined in some belt-shaped regions, or *thick* – enjoying good pumping properties that can be exploited to shrink long runs. We extend this technique and prove a similar thin-thick classification for runs in geometrically 2-dimensional VASSes. This will enable us to obtain the following exponential length bound for reachability witnesses.

▶ **Theorem 4.1.** *For any* $\mathbf{0}$*-run* τ *in a geometrically 2-dimensional* d*-VASS* G*, there is a* $\mathbf{0}$*-run* ρ *in* G *with the same source and target states as* τ *and* $|\rho| \leq \chi(G)^{O(\varsigma(G)\cdot d^4)}$*.*

From the above theorem the PSPACE-completeness follows immediately.

▶ **Theorem 4.2.** *Reachability in geometrically 2-dimensional VASS is* PSPACE*-complete.*

Proof. The lower bound is inherited from the PSPACE-hardness of reachability in 2-VASS [3, Lemma 20]. For the upper bound, an algorithm only need to search a run of length up to $\chi(G)^{O(\varsigma(G)\cdot d^4)}$ after reducing to the $\mathbf{0}$-reachability problem, for which PSPACE is enough. ◀

In order to establish the thin-thick classification of runs in a geometrically 2-dimensinoal VASS, we need to create a suitable coordinate system within its cycle space. This is achieved using the projection tools developed in Subsection 4.1 and Subsection 4.2. Depending on whether these tools can be applied, we classify geometrically 2-dimensinoal VASSes into degenerate ones (where projection tools are not applicable) and proper ones (where projection tools are applicable). Degenerate VASSes are easier to handle as they only admit thin runs, see Subsection 4.3. For proper VASSes in Subsection 4.4 we adapt the argument of [6] within the coordinate system created by the projection tools and establish the thin-thick classification. Finally we recall in Subsection 4.5 that both thin runs and thick runs can be shrunk to exponential length, thus Theorem 4.1 follows.

4.1 Sign Reflecting Projection

An *orthant* is one of the 2^d regions in $\mathbb{Q}^d$ split by the d axes. Formally, given a vector $t \in \{+1, -1\}^d$, the orthant Z_t defined by t is the set $Z_t := \{\boldsymbol{u} \in \mathbb{Q}^d : \boldsymbol{u}(i) \cdot t(i) \geq 0 \text{ for all } i \in [d]\}$. The non-negative orthant $\mathbb{Q}^d_{\geq 0} = Z_{(+1,+1,\ldots,+1)}$ is a major concern in this paper.

Let $I \subseteq [d]$ be a subset of indices. For a vector $\boldsymbol{u} \in \mathbb{Q}^d$, we define its *projection onto indices in* I as a function $\boldsymbol{u}|_I \in \mathbb{Q}^I$ given by $(\boldsymbol{u}|_I)(i) = \boldsymbol{u}(i)$ for all $i \in I$. We tacitly identify the function $\boldsymbol{u}|_I \in \mathbb{Q}^I$ as a vector in $\mathbb{Q}^{|I|}$. For a set of vectors $V \subseteq \mathbb{Q}^d$, we define $V|_I := \{\boldsymbol{v}|_I : \boldsymbol{v} \in V\}$. It should be clear that the projection of a vector space onto indices in I is again a vector space in $\mathbb{Q}^{|I|}$, and the projection of an orthant Z_t onto I is an orthant in $\mathbb{Q}^{|I|}$ defined by $t|_I$.

▶ **Definition 4.3** ([12, Definition A.7])**.** *Let* $P \subseteq \mathbb{Q}^d$ *be a vector space and* Z *be an orthant in* $\mathbb{Q}^d$*. A set of indices* $I \subseteq [d]$ *is called a* sign-reflecting projection for P with respect to Z *if for any* $\boldsymbol{v} \in P$*,* $\boldsymbol{v}|_I \in Z|_I$ *implies* $\boldsymbol{v} \in Z$*.*

Sign-reflecting projection helps us project the vectors in a vector space to some of its components so that the pre-image of a certain orthant still belongs to one orthant. Moreover, we have that such a projection is one-to-one.

▶ **Lemma 4.4** ([12, Lemma A.8])**.** *Let* $P \subseteq \mathbb{Q}^d$ *be a vector space and* Z *be an orthant in* $\mathbb{Q}^d$*. Let* $I \subseteq [d]$ *be a sign-reflecting projection for* P *w.r.t.* Z*. Then every vector* $\boldsymbol{v} \in P$ *is uniquely determined by* $\boldsymbol{v}|_I$*. In other words, for any* $\boldsymbol{v}, \boldsymbol{v}' \in P$*,* $\boldsymbol{v}|_I = \boldsymbol{v}'|_I$ *implies* $\boldsymbol{v} = \boldsymbol{v}'$*.*

We mainly care about sign-reflecting projections for a plane, i.e. a 2-dimensional subspace of $\mathbb{Q}^d$. In this case a good sign-reflecting projection is given by the following lemma.

▶ **Lemma 4.5** ([12, Theorem A.9])**.** *For* $d \geq 2$*, let* $P \subseteq \mathbb{Q}^d$ *be a plane (i.e. a 2-dimensional subspace), and* Z *be an orthant in* $\mathbb{Q}^d$ *such that* $P \cap Z$ *contains two linearly independent vectors. Then there is a sign-reflecting projection* I *for* P *w.r.t.* Z *such that* $|I| = 2$*.*

Intuitively Lemma 4.5 projects a plane onto an axis plane. The preimages of those two axes will play an important role in our work, as stated in the following lemma.

▶ **Lemma 4.6.** *Let $P \subseteq \mathbb{Q}^d$ be a plane. Let $I = \{i_1, i_2\} \subseteq [d]$ be a sign-reflecting projection for P w.r.t. $\mathbb{Q}^d_{\geq 0}$. Suppose $P = \mathrm{span}\{\boldsymbol{v}_1, \boldsymbol{v}_2\}$ for linearly independent vectors $\boldsymbol{v}_1, \boldsymbol{v}_2 \in \mathbb{Z}^d$ with norm $N := \max\{\|\boldsymbol{v}_1\|, \|\boldsymbol{v}_2\|\}$. Then there exists two non-zero vectors $\boldsymbol{u}_1, \boldsymbol{u}_2 \in P \cap \mathbb{N}^d$ such that $\boldsymbol{u}_1(i_2) = \boldsymbol{u}_2(i_1) = 0$ and $\boldsymbol{u}_1(i_1) = \boldsymbol{u}_2(i_2) > 0$ and that $\|\boldsymbol{u}_1\|, \|\boldsymbol{u}_2\| \leq 2N^2$.*

We call vectors $\boldsymbol{u}_1, \boldsymbol{u}_2$ given by this lemma the *canonical horizontal / vertical vector* derived from $\boldsymbol{v}_1, \boldsymbol{v}_2$ for P with respect to $\mathbb{Q}^d_{\geq 0}$. We remark that this notion can be generalized to orthants other than $\mathbb{Q}^d_{\geq 0}$. As any vector in P is uniquely represented as a linear combination of $\boldsymbol{u}_1$ and $\boldsymbol{u}_2$, we can obtain a bound for components of vectors in P in terms of their projections.

▶ **Lemma 4.7.** *Let $P \subseteq \mathbb{Q}^d$ be a plane. Let $I = \{i_1, i_2\} \subseteq [d]$ be a sign-reflecting projection for P w.r.t. $\mathbb{Q}^d_{\geq 0}$. Suppose $P = \mathrm{span}\{\boldsymbol{v}_1, \boldsymbol{v}_2\}$ for linearly independent vectors $\boldsymbol{v}_1, \boldsymbol{v}_2 \in \mathbb{Z}^d$ with norm $N := \max\{\|\boldsymbol{v}_1\|, \|\boldsymbol{v}_2\|\}$. For every $\boldsymbol{w} \in P \cap \mathbb{N}^d$ and every $i \in \mathrm{supp}(P)$, we have*

$$\frac{\min\{\boldsymbol{w}(i_1), \boldsymbol{w}(i_2)\}}{2N^2} \leq \boldsymbol{w}(i) \leq 2N^2 \cdot (\boldsymbol{w}(i_1) + \boldsymbol{w}(i_2)). \tag{5}$$

4.2 Support Projection

Lemma 4.7 gives bounds on the components in the support of the plane. So we would like the cycle space of a VASS G to have full support, i.e. $\mathrm{supp}(\mathrm{Cyc}(G)) = [d]$. In this section we develop a technique called *support projection* to transform an arbitrary geometrically 2-dimensional VASS to one with such good property, without increasing its traversal number and the characteristic.

Let $G = (Q, T)$ be a geometrically 2-dimensional d-VASS. Let $S = \mathrm{supp}(\mathrm{Cyc}(G))$ and $\overline{S} = [d] \setminus S$. For vectors $\boldsymbol{v} \in \mathbb{Z}^S$ and $\overline{\boldsymbol{v}} \in \mathbb{Z}^{\overline{S}}$, we define their composition $\boldsymbol{v} \circ_S \overline{\boldsymbol{v}} \in \mathbb{Z}^d$ naturally by $(\boldsymbol{v} \circ_S \overline{\boldsymbol{v}})(i) = \boldsymbol{v}(i)$ if $i \in S$ and $(\boldsymbol{v} \circ_S \overline{\boldsymbol{v}})(i) = \overline{\boldsymbol{v}}(i)$ if $i \in \overline{S}$. Since S is always clear from the context, we will simply write $\boldsymbol{v} \circ \overline{\boldsymbol{v}}$ for this composition.

The *support projection* of G is the $|S|$-dimensional VASS $G^S = (Q^S, T^S)$ where

$$Q^S := \{(q, \boldsymbol{v}) \in Q \times \mathbb{N}^{\overline{S}} : \|\boldsymbol{v}\| \leq 2\chi(G)\}, \tag{6}$$

$$T^S := \{((p, \boldsymbol{u}), \boldsymbol{a}|_S, (q, \boldsymbol{v})) \in Q^S \times \mathbb{N}^S \times Q^S : (p, \boldsymbol{a}, q) \in T, \boldsymbol{u} + \boldsymbol{a}|_{\overline{S}} = \boldsymbol{v}\}. \tag{7}$$

A state of the form $(q, \boldsymbol{v})$ in G^S is denoted $q^{\boldsymbol{v}}$ for conciseness.

There is a huge expansion in the size of G^S, as $|Q^S| = |Q| \cdot (2\chi(G))^{|\overline{S}|}$. On the other hand, we can show that support projection does not increase traversal number and characteristic, and the projected VASS has full support as we expected.

▶ **Proposition 4.8.** *$\varsigma(G^S) \leq \varsigma(G)$, $\chi(G^S) \leq \chi(G)$, and $\mathrm{supp}(\mathrm{Cyc}(G^S)) = S$.*

Proof. Let's denote by $\varsigma(\pi)$ the number of distinct states in the path π. Consider any path π^S in G^S of the form $\pi^S = p_0^{\boldsymbol{v}_0} \xrightarrow{t_1^S} p_1^{\boldsymbol{v}_1} \xrightarrow{t_2^S} \cdots \xrightarrow{t_n^S} p_n^{\boldsymbol{v}_n}$ where $t_i^S \in T^S$. We define a corresponding path π in G as $\pi := p_0 \xrightarrow{t_1} p_1 \xrightarrow{t_2} \cdots \xrightarrow{t_n} p_n$ where $t_i = (p_{i-1}, \boldsymbol{a}_i, p_i)$ and $\boldsymbol{a}_i := \Delta(t_i^S) \circ (\boldsymbol{v}_i - \boldsymbol{v}_{i-1})$. Verify that t_i is indeed a transition in T by the definition of T^S. We claim that for any $i, j \in [0, n]$, $p_i = p_j$ implies $\boldsymbol{v}_i = \boldsymbol{v}_j$, then it follows that $\varsigma(\pi^S) = \varsigma(\pi)$. Indeed, suppose $p_i = p_j$, then the sub path in π from p_i to p_j is a cycle with effect $\Delta(t_{i+1}t_{i+2}\dots t_j) \in \mathrm{Cyc}(G)$. So $\boldsymbol{v}_j = \boldsymbol{v}_i + \Delta(t_{i+1}t_{i+2}\dots t_j)|_{\overline{S}} = \boldsymbol{v}_i$, which proves the claim. As the choice of π^S is arbitrary, for any path in G^S, there exists a path in G visits the same number of distinct states. This proves $\varsigma(G) \geq \varsigma(G^S)$.

Since it is clear that $\|T^S\| \leq \|T\|$, we immediately have $\chi(G^S) = \varsigma(G^S) \cdot \|T^S\| \leq \varsigma(G) \cdot \|T\| = \chi(G)$.

Finally, we show $\mathrm{supp}(\mathrm{Cyc}(G^S)) = S$. Observe that it suffices to prove $\mathrm{Cyc}(G)|_S \subseteq \mathrm{Cyc}(G^S)$. By Lemma 3.2, $\mathrm{Cyc}(G)$ is spanned by effects of all simple cycles in G. So consider any cycle θ in G of the form $\theta := p_0 \xrightarrow{t_1} p_1 \xrightarrow{t_2} \cdots \xrightarrow{t_n} p_n = p_0$ where $t_1, \ldots, t_n \in T$. We define vectors $\boldsymbol{v}_0, \ldots, \boldsymbol{v}_n \in \mathbb{N}^{\overline{S}}$ by

$$\boldsymbol{v}_0 := \chi(G) \cdot \mathbf{1}, \qquad \boldsymbol{v}_{i+1} := \boldsymbol{v}_i + \Delta(t_{i+1})|_{\overline{S}}. \tag{8}$$

Since θ is simple, we have $n \leq \varsigma(G)$. It follows that $\mathbf{0} \leq \boldsymbol{v}_i \leq 2\chi(G) \cdot \mathbf{1}$ for all $i \in [0, n]$. So $p_i^{\boldsymbol{v}_i}$ is a state in G^S. Also note that $\boldsymbol{v}_n = \boldsymbol{v}_0 + \Delta(\theta)|_{\overline{S}} = \boldsymbol{v}_0$. We can define a corresponding cycle $\theta^S := p_0^{\boldsymbol{v}_0} \xrightarrow{t_1^S} p_1^{\boldsymbol{v}_1} \xrightarrow{t_2^S} \cdots \xrightarrow{t_n^S} p_n^{\boldsymbol{v}_n}$, where $t_i = (p_{i-1}^{\boldsymbol{v}_{i-1}}, \Delta(t_i)|_S, p_i^{\boldsymbol{v}_i})$. Verify that each t_i is a transition in T^S by definition. So θ^S is a cycle in G^S. In particular, we have $\Delta(\theta)|_S = \Delta(\theta^S) \in \mathrm{Cyc}(G^S)$. As the choice of θ is arbitrary, we conclude that

$$\mathrm{Cyc}(G)|_S = (\mathrm{span}\{\Delta(\theta) : \theta \text{ is a simple cycle in } G\})|_S \subseteq \mathrm{Cyc}(G^S), \tag{9}$$

which is the desired result. ◀

Using the support projection we can change our focus on VASSes with full support. We will prove the following lemma in the remaining of this section.

▶ **Lemma 4.9.** *For any* $\mathbf{0}$*-run* τ *in a geometrically 2-dimensional* d*-VASS* G *with the additional property that* $\mathrm{supp}(\mathrm{Cyc}(G)) = [d]$*, there is a* $\mathbf{0}$*-run* ρ *in* G *with the same source and target states as* τ *and* $|\rho| \leq \chi(G)^{O(\varsigma(G) \cdot d^4)}$*.*

Once Lemma 4.9 is established, Theorem 4.1 follows by plugging in the support projection of G. The rest is devoted to Lemma 4.9, so we can always assume that the VASS G has full support, i.e., $\mathrm{supp}(\mathrm{Cyc}(G)) = [d]$.

4.3 Degenerate VASS and Thin Runs

Depending on whether the cycle space of a VASS can be sign-reflectively projected onto an axes-plane with respect to the non-negative orthant $\mathbb{Q}^d_{\geq 0}$, we classify geometrically 2-dimensional VASSes into the following two classes.

▶ **Definition 4.10.** *A geometrically 2-dimensional VASS* G *is* proper *if* $\mathrm{Cyc}(G) \cap \mathbb{Q}^d_{\geq 0}$ *contains two linearly independent vectors; it is* degenerate *otherwise.*

In this subsection we focus on degenerate VASSes. We show that every run from $\mathbf{0}$ in a degenerate VASS is *thin* in the sense of the following definitions. An illustration of thin runs can be found in Figure 1a.

▶ **Definition 4.11.** *Let* $\boldsymbol{v} \in \mathbb{N}^d$ *and* $W \in \mathbb{N}$*. The* beam $\mathcal{B}_{\boldsymbol{v},W}$ *is defined by*

$$\mathcal{B}_{\boldsymbol{v},W} := \left\{\boldsymbol{u} \in \mathbb{N}^d : \exists \alpha \in \mathbb{Q}_{\geq 0}, \|\boldsymbol{u} - \alpha \boldsymbol{v}\| \leq W\right\}. \tag{10}$$

The beam $\mathcal{B}_{\boldsymbol{v},W}$ *is said to be an* A-beam *where* $A \in \mathbb{N}$ *if* $\|\boldsymbol{v}\| \leq A$ *and* $W \leq A$*.*

▶ **Definition 4.12.** *Let* G *be a* d*-VASS. A run* π *in* G *is said to be* A-thin *if for every configuration* $p(\boldsymbol{u})$ *occurring in* π*, the vector* $\boldsymbol{u}$ *belongs to some* A*-beam.*

Indeed, we can relax the definition of beams by letting the direction $\boldsymbol{v}$ range over all integer vectors in $\mathbb{Z}^d$. Let $\boldsymbol{v} \in \mathbb{Z}^d$ and $W \in \mathbb{N}$. The *generalized beam* $\mathcal{B}^{\mathbb{Z}}_{\boldsymbol{v},W}$ is defined by

$$\mathcal{B}^{\mathbb{Z}}_{\boldsymbol{v},W} := \left\{\boldsymbol{u} \in \mathbb{N}^d : \exists \alpha \in \mathbb{Q}, \|\boldsymbol{u} - \alpha \boldsymbol{v}\| \leq W\right\}. \tag{11}$$

▶ **Lemma 4.13.** *For any $\boldsymbol{v} \in \mathbb{Z}^d$ and $W \in \mathbb{N}$, there exist $\boldsymbol{v}^+ \in \mathbb{N}^d$ and $\boldsymbol{v}^- \in \mathbb{N}^d$ such that $\mathcal{B}^{\mathbb{Z}}_{\boldsymbol{v},W} \subseteq \mathcal{B}_{\boldsymbol{v}^+,W} \cup \mathcal{B}_{\boldsymbol{v}^-,W}$ and that $\|\boldsymbol{v}^+\|, \|\boldsymbol{v}^-\| \leq \|\boldsymbol{v}\|$.*

With Lemma 4.13, Definition 4.12 is equivalent to stating that each configuration is located in some generalized A-beam. This makes it easier to argue if a run is thin, as demonstrated in the following lemma. We remark that the following result does not depend on whether the VASS has full support.

▶ **Lemma 4.14.** *Let G be a geometrically 2-dimensional VASS that is degenerate. Then every $\mathbf{0}$-run in G is $\chi(G)^{O(d)}$-thin.*

Proof sketch. A degenerate VASS G falls into one of the following 3 cases: (i) $\mathrm{gdim}(G) < 2$; (ii) $\mathrm{gdim}(G) = 2$ and $\mathrm{Cyc}(G) \cap \mathbb{Q}^d_{\geq 0} = \{\mathbf{0}\}$, or (iii) $\mathrm{gdim}(G) = 2$ and $\mathrm{Cyc}(G) \cap \mathbb{Q}^d_{\geq 0} = \mathbb{Q}_{\geq 0} \cdot \boldsymbol{u}$ for some $\boldsymbol{u} \in \mathbb{N}^d \setminus \{\mathbf{0}\}$. We consider only the first case here. For the other cases we refer the readers to the full version of this paper.

Suppose $\mathrm{gdim}(G) < 2$, then $\mathrm{Cyc}(G) = \mathrm{span}\{\boldsymbol{c}\}$ where $\boldsymbol{c}$ is the effect of a (possibly empty) simple cycle in G. So $\|\boldsymbol{c}\| \leq \chi(G)$. By Lemma 3.4 every configuration $q(\boldsymbol{v})$ reachable from $p(\mathbf{0})$ satisfy $\boldsymbol{v} = \alpha \boldsymbol{c} + \boldsymbol{z}$ for some $\alpha \in \mathbb{Q}$ and $\|\boldsymbol{z}\| \leq \chi(G)$. This shows that $\boldsymbol{v} \in \mathcal{B}^{\mathbb{Z}}_{\boldsymbol{c},\chi(G)}$. As the choice of $q(\boldsymbol{v})$ is arbitrary, we deduce that every $\mathbf{0}$-run in G is confined in the generalized beam $\mathcal{B}^{\mathbb{Z}}_{\boldsymbol{c},\chi(G)}$, thus is $\chi(G)$-thin. ◀

4.4 Proper VASS and the Thin-Thick Classification

In this subsection we fix a geometrically 2-dimensional d-VASS $G = (Q, T)$ that is proper, and assume that $\mathrm{supp}(\mathrm{Cyc}(G)) = [d]$. So by Lemma 4.5, there exists $i_1 \neq i_2 \in [d]$ such that $I := \{i_1, i_2\}$ is a sign-reflecting projection of $\mathrm{Cyc}(G)$ with respect to $\mathbb{Q}^d_{\geq 0}$. Moreover, let $\boldsymbol{u}_1, \boldsymbol{u}_2 \in \mathbb{N}^d$ be the canonical horizontal and vertical vectors given by Lemma 4.6. We have $\boldsymbol{u}_1(i_2) = 0$ and $\boldsymbol{u}_2(i_1) = 0$. Observe that we can assume $\|\boldsymbol{u}_1\|, \|\boldsymbol{u}_2\| \leq 2\chi(G)^2$, as they can be derived from effects of simple cycles in G.

4.4.1 Thick Runs

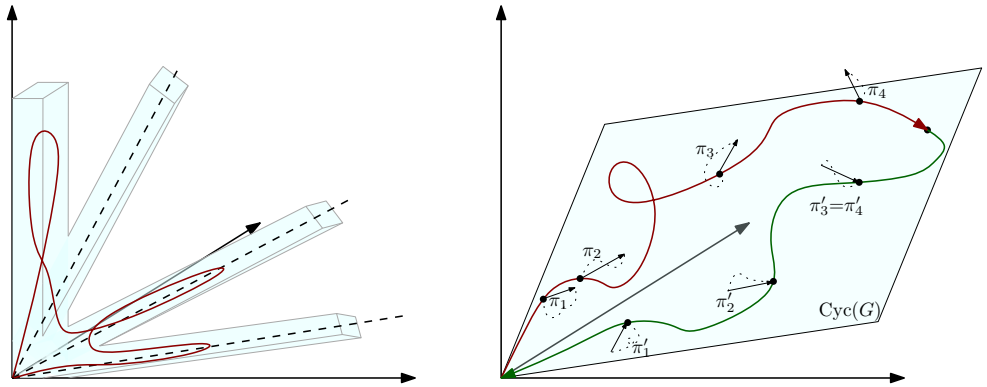
We will show that every $\mathbf{0}$-run in G can be classified into thin runs and thick runs. Here we give the definition of thick runs.

Sequential cones

Recall that the *cone* generated by vectors $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_k \in \mathbb{Z}^d$ is the set $\mathrm{Cone}\{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_k\} := \{\sum_{j=1}^k a_j \boldsymbol{v}_j : a_1, \ldots, a_k \in \mathbb{Q}_{\geq 0}\}$. The definition of cones is enhanced in [6] where every prefix sum is also required to be non-negative:

▶ **Definition 4.15** (sequential cones)**.** *Let $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_{\boldsymbol{k}} \in \mathbb{Z}^d$ be a sequence of vectors, the* sequential cone *generated by these vectors is the following set:*

$$\mathrm{SeqCone}(\boldsymbol{v}_1, \ldots, \boldsymbol{v}_{\boldsymbol{k}}) := \left\{\sum_{j=1}^k a_j \boldsymbol{v}_j : a_1, \ldots, a_k \in \mathbb{Q}_{\geq 0}, \forall i. \sum_{j=1}^i a_j \boldsymbol{v}_j \geq \mathbf{0}\right\}. \tag{12}$$



(a) Thin runs. **(b)** Thick runs.

Figure 1 Illustration of thin runs and thick runs.

In dimension 2 it was shown that a sequential cone is nothing but a cone generated by 2 vectors [6]. We generalize this result to sequential cones generated by vectors from the cycle space of a proper geometrically 2-dimensional VASS.

▶ **Lemma 4.16.** *Let* $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_k \in \mathbb{Z}^d$ *be vectors in* $\mathrm{Cyc}(G)$*. Then* $\mathrm{SeqCone}(\boldsymbol{v}_1, \ldots, \boldsymbol{v}_k) = \mathrm{Cone}\{\boldsymbol{x}, \boldsymbol{y}\}$ *for some non-negative vectors* $\boldsymbol{x}, \boldsymbol{y}$ *where each of them is either* $\boldsymbol{v}_j$ *for some* $j \in [k]$*, or is the canonical horizontal / vertical vector* $\boldsymbol{u}_1$ *or* $\boldsymbol{u}_2$*.*

Indeed, this lemma reduces to [6, Lemma 2] easily by projecting the sequential cone onto coordinates in I.

Sequentially enabled cycles and thick runs

A path π is *enabled* at configuration c if there exists a configuration c' such that $c \xrightarrow{\pi} c'$ is a legal run. Let $S \subseteq [d]$, we say π is S-*enabled* at $c = p(\boldsymbol{u})$ if there exists a vector $\boldsymbol{z} \in \mathbb{N}^d$ with $\mathrm{supp}(\boldsymbol{z}) \subseteq [d] \setminus S$ such that π is enabled at $p(\boldsymbol{u} + \boldsymbol{z})$. In other words, π is S-enabled if it is enabled when ignoring the coordinates outside S.

▶ **Definition 4.17.** *Let* $A \in \mathbb{N}$*, and let* $\pi_1, \pi_2, \pi_3, \pi_4$ *be four cycles in* G*, we say these cycles are* A-sequentially enabled *in a run* ρ *in* G *if their lengths are at most* A*, and* ρ *can be factored into five parts* $\rho = \rho_1\rho_2\rho_3\rho_4\rho_5$ *such that*

- $\Delta(\pi_1)|_I$ *is semi-positive, and* π_1 *is enabled at* $\mathrm{trg}(\rho_1)$*. Moreover, both coordinates in* I *are bounded by* A *along* ρ_1*.*
- *If* $\Delta(\pi_1)|_I$ *is positive, then* π_2 *is* $\emptyset$*-enabled at* $\mathrm{trg}(\rho_2)$*. Otherwise,* π_2 *is* S*-enabled at* $\mathrm{trg}(\rho_2)$ *for* $S = [d] \setminus \mathrm{supp}(\Delta(\pi_1))$*, and, if* $\Delta(\pi_1)(i_b) = 0$*, then the* i_b*-th coordinate is bounded by* A *along* ρ_2 *where* $b = 1, 2$*.*
- $\mathrm{SeqCone}(\Delta(\pi_1), \Delta(\pi_2))$ *contains some positive vector. (We remark that this is possible only if* G *has full support.)*
- π_3, π_4 *are* $\emptyset$*-enabled at* $\mathrm{trg}(\rho_3), \mathrm{trg}(\rho_4)$ *respectively.*

▶ **Definition 4.18.** *Let* $A \in \mathbb{N}$*. A* **0***-run* τ *in* G *is* A-thick *if* τ *factors into* $\tau = \rho\rho'$ *such that*

- *some cycles* $\pi_1, \pi_2, \pi_3, \pi_4$ *in* G *are* A*-sequentially enabled in* ρ*,*
- *some cycles* $\pi_1', \pi_2', \pi_3', \pi_4'$ *in* G^{rev} *are* A*-sequentially enabled in* $\mathrm{rev}(\rho)$*,*
- $\mathrm{SeqCone}(\Delta(\pi_1), \Delta(\pi_2), \Delta(\pi_3), \Delta(\pi_4)) \cap \mathrm{SeqCone}(\Delta(\pi_1'), \Delta(\pi_2'), \Delta(\pi_3'), \Delta(\pi_4'))$ *is not trivial (i.e. it contains two linearly independent vectors).*

An illustration of thick runs is given in Figure 1b.

4.4.2 Thin-Thick Classification

In spirit of [6], the following classification lemma is of great significance.

▶ **Lemma 4.19.** *Let G be a proper geometrically 2-dimensional d-VASS with $\mathrm{supp}(\mathrm{Cyc}(G)) = [d]$, then there exists a number $A \leq \chi(G)^{O(\varsigma(G))}$ such that every $\mathbf{0}$-run in G is A-thick if it is not A-thin.*

An important technical lemma in [6] is the "non-negative cycle lemma" [6, Lemma 3], which states that a run in 2-VASS from $\mathbf{0}$ visiting a high configuration must contain a configuration enabling a semi-positive cycle. Here we need a similar lemma for geometrically 2-dimensional VASS.

▶ **Lemma 4.20.** *There exists a polynomial P such that every run ρ in G from $\mathbf{0}$ to $\boldsymbol{v}$ with $\|\boldsymbol{v}\| \geq P(\chi(G))^{\varsigma(G)}$ contains a configuration enabling a cycle θ of length at most $P(\chi(G))$ such that $\Delta(\theta)|_I$ is semi-positive.*

We also need some simple geometric facts.

▶ **Lemma 4.21.** *Let $\boldsymbol{u}, \boldsymbol{v} \in \mathbb{Q}^d$. Let $X \subseteq \mathrm{span}\{\boldsymbol{u}, \boldsymbol{v}\}$ be a convex set such that $X \cap \mathbb{Q}_{\geq 0} \cdot \boldsymbol{u} = \emptyset$ and $X \cap \mathbb{Q}_{\geq 0} \cdot \boldsymbol{v} = \emptyset$. Then either $X \cap \mathrm{Cone}\{\boldsymbol{u}, \boldsymbol{v}\} = \emptyset$ or $X \subseteq \mathrm{Cone}\{\boldsymbol{u}, \boldsymbol{v}\}$.*

Given a 2-vector $\boldsymbol{v} \in \mathbb{Q}^2$, we define its *right rotation* $\boldsymbol{v}^R := (\boldsymbol{v}(2), -\boldsymbol{v}(1))$. For another vector $\boldsymbol{u} \in \mathbb{Q}^2$, we write $\boldsymbol{v} \curvearrowright \boldsymbol{u}$ if $\langle \boldsymbol{u}, \boldsymbol{v}^R \rangle > 0$, and write $\boldsymbol{v} \underset{\frown}{\curvearrowright} \boldsymbol{u}$ if $\langle \boldsymbol{u}, \boldsymbol{v}^R \rangle \geq 0$. We generalize this notation to the 2-dimensional subspace $\mathrm{Cyc}(G)$: given two vectors $\boldsymbol{u}, \boldsymbol{v} \in \mathrm{Cyc}(G)$, we write $\boldsymbol{v} \curvearrowright \boldsymbol{u}$ if $\boldsymbol{v}|_I \curvearrowright \boldsymbol{u}|_I$, and $\boldsymbol{v} \underset{\frown}{\curvearrowright} \boldsymbol{u}$ if $\boldsymbol{v}|_I \underset{\frown}{\curvearrowright} \boldsymbol{u}|_I$.

▶ **Proposition 4.22.** *Let $\boldsymbol{u}, \boldsymbol{v} \in \mathrm{Cyc}(G)$ be such that $\boldsymbol{u} \curvearrowright \boldsymbol{v}$. Then for any $\boldsymbol{w} \in \mathrm{Cyc}(G)$, $\boldsymbol{w} \in \mathrm{Cone}\{\boldsymbol{u}, \boldsymbol{v}\}$ if and only if $\boldsymbol{u} \underset{\frown}{\curvearrowright} \boldsymbol{w} \underset{\frown}{\curvearrowright} \boldsymbol{v}$.*

The threshold A

Let P be the polynomial in Lemma 4.20. We define p to be the polynomial

$$p(x) = 4x^4 \cdot (2x^3 \cdot (P(x) + (x+1)^2 + 5) + 2x) + x. \tag{13}$$

Define $A := p(\chi(G))^{\varsigma(G)} \leq \chi(G)^{O(\varsigma(G))}$. Note that in particular we have

$$A \geq p(\chi(G))^{\varsigma(G)} \geq 4\chi(G)^4 \cdot (B + \chi(G)) + \chi(G) \tag{14}$$

where $B := 2\chi(G)^3 \cdot (P(\chi(G))^{\varsigma(G)} + (\chi(G)+1)^2 + 5) + \chi(G)$.

If a $\mathbf{0}$-run is not A-thin, then we can find a configuration that lies out of all A-beams. The property that G has full support helps to further show that each component of this configuration is high.

▶ **Lemma 4.23.** *Let ρ be a $\mathbf{0}$-run in G that is not A-thin. Then ρ contains a configuration $s(\boldsymbol{w})$ where $\boldsymbol{w}$ lies outside all A-beams, and such that $\boldsymbol{w}(i) \geq B$ for all $i \in [d]$.*

Main Lemma

▶ **Lemma 4.24.** *Let τ be a $\mathbf{0}$-run in G that is not A-thin. Let $s(\boldsymbol{w})$ be the configuration on τ given by Lemma 4.23. Then τ can be factored into two parts $\tau = \rho\rho'$ where $\mathrm{trg}(\rho) = s(\boldsymbol{w}) = \mathrm{src}(\rho')$ such that*

- *There are 4 cycles $\pi_1, \pi_2, \pi_3, \pi_4$ in G that are B-sequentially enabled in ρ, such that the set* $\mathrm{SeqCone}(\Delta(\pi_1), \Delta(\pi_2), \Delta(\pi_3), \Delta(\pi_4))$ *contains a vector $\boldsymbol{x}$ with $\|\boldsymbol{x} - \boldsymbol{w}\| \leq \chi(G)$.*
- *There are 4 cycles $\pi'_1, \pi'_2, \pi'_3, \pi'_4$ in G^{rev} that are B-sequentially enabled in $rev(\rho')$, such that* $\mathrm{SeqCone}(\Delta(\pi'_1), \Delta(\pi'_2), \Delta(\pi'_3), \Delta(\pi'_4))$ *contains a vector $\boldsymbol{x}'$ with $\|\boldsymbol{x}' - \boldsymbol{w}\| \leq \chi(G)$.*

We show that Lemma 4.19 follows immediately from Lemma 4.24.

Proof of Lemma 4.19. If the run τ is not A-thin, then Lemma 4.24 applies. Regarding the definition of thick runs, we are only left to show $\mathrm{SeqCone}(\Delta(\pi_1), \Delta(\pi_2), \Delta(\pi_3), \Delta(\pi_4)) \cap \mathrm{SeqCone}(\Delta(\pi'_1), \Delta(\pi'_2), \Delta(\pi'_3), \Delta(\pi'_4))$ contains two linearly independent vectors. Define $U := \{\boldsymbol{u} \in \mathrm{Cyc}(G) : \|\boldsymbol{u} - \boldsymbol{w}\| \leq 2 \cdot \chi(G)\}$. By Lemma 4.16, $\mathrm{SeqCone}(\Delta(\pi_1), \Delta(\pi_2), \Delta(\pi_3), \Delta(\pi_4))$ is equal to some $\mathrm{Cone}\{\boldsymbol{v}_1, \boldsymbol{v}_2\}$ where $\|\boldsymbol{v}_1\|, \|\boldsymbol{v}_2\| \leq B \cdot \chi(G) \leq A$. As $\boldsymbol{w}$ lies out of all A-beams, we must have $U \cap \mathbb{Q}_{\geq 0}\boldsymbol{v}_1 = \emptyset$ and $U \cap \mathbb{Q}_{\geq 0}\boldsymbol{v}_2 = \emptyset$. On the other hand, Lemma 4.24 guarantees that $\mathrm{SeqCone}(\Delta(\pi_1), \Delta(\pi_2), \Delta(\pi_3), \Delta(\pi_4)) \cap U \neq \emptyset$. So by Lemma 4.21, we have $U \subseteq \mathrm{SeqCone}(\Delta(\pi_1), \Delta(\pi_2), \Delta(\pi_3), \Delta(\pi_4))$. A similar argument shows also that $U \subseteq \mathrm{SeqCone}(\Delta(\pi'_1), \Delta(\pi'_2), \Delta(\pi'_3), \Delta(\pi'_4))$. Finally, one easily verifies that U contains two linearly independent vectors. ◀

Proof sketch of Lemma 4.24. By symmetry we only need to prove the first item. Since ρ reaches the configuration $s(\boldsymbol{w})$ which is high enough, Lemma 4.20 shows that ρ contains a configuration c_1 enabling a semi-positive cycle π_1. For the second cycle, if π_1 is positive, then we simply let $\pi_2 := \pi_1$. Otherwise, $\Delta(\pi_1)$ is parallel to one of the canonical vectors $\boldsymbol{u}_1$ or $\boldsymbol{u}_2$. Say it is the latter case, so $\Delta(\pi_1)(i_1) = 0$. We need to find a cycle π_2 with $\Delta(\pi_2)(i_1) > 0$. First observe that Lemma 4.20 allows us to assume $\|c_1\| \leq P(\chi(G))^{\varsigma(G)} + \chi(G)$. As $\boldsymbol{w}(i_1) \geq B > c_i(i_1) + \chi(G)$, the run from c_1 to $s(\boldsymbol{w})$ must contain a cycle with positive effect in i_1-th coordinate. We would like to choose this cycle to be π_2, but its length may be unbounded. So we remove from it all sub-cycles whose effect is in parallel with $\Delta(\pi_1)$. This makes π_2 merely $([d] \setminus \mathrm{supp}(\Delta(\pi_1)))$-enabled, but that's enough. Finally, we exhibit cycles π_3, π_4 so that the sequential cone contains some vector $\boldsymbol{x}$ in the neighbor of $\boldsymbol{w}$. We would like $\boldsymbol{x}$ to be exactly $\boldsymbol{w}$, but this may be impossible as $\boldsymbol{w}$ might not even belong to $\mathrm{Cyc}(G)$. So we write $\boldsymbol{w} = \boldsymbol{c} + \boldsymbol{z}$ for $\boldsymbol{c} \in \mathrm{Cyc}(G)$ and $\|\boldsymbol{z}\| \leq \chi(G)$. The goal is now to contain $\boldsymbol{x} = \boldsymbol{c}$ in the sequential cone. Consider all simple cycles enabled at some configuration on the run ρ after the source of π_2. List them in the order they are enabled, and extract a subsequence $\pi_3, \pi_4, \ldots$ such that $\Delta(\pi_2) \curvearrowright \Delta(\pi_3) \curvearrowright \Delta(\pi_4) \curvearrowright \cdots$. So the effects of them sweep the cycle space clockwise. Then we are able to argue, as in [6], that there must be some π_i, π_{i+1} in this sequence such that $\Delta(\pi_i) \curvearrowright \boldsymbol{c} \curvearrowright \Delta(\pi_{i+1})$. We are done by choosing them as π_3 and π_4. ◀

4.5 Exponential Bounds of Reachability Witnesses

Lemma 4.19, together with Lemma 4.14, shows that every run in a geometrically 2-dimensional VASS G is either A-thin or A-thick for $A = \chi(G)^{O(d \cdot \varsigma(G))}$. For both cases we are able to exhibit a reachability witness of length exponential in A. The proofs are similar to that in [6]. So here we only state the results with basic sketches of the proof.

▶ **Lemma 4.25.** *For any $\mathbf{0}$-run τ that is A-thin in a d-VASS G where $A \geq 2\chi(G)$, there exists a $\mathbf{0}$-run ρ with the same source and target states as τ such that $|\rho| \leq A^{O(d^2)}$.*

Proof sketch. Recall that A-thin runs are confined in A-beams of finitely many possibilities. These beams become eventually disjoint in the region far away from the origin. If an A-thin $\mathbf{0}$-run τ without repeated configuration is long enough, it must contain a configuration c whose vector is far from origin. In this case we can find a sub-run π around c that stays

within a unique A-beam. As τ is a $\mathbf{0}$-run, π first goes "up" and then "down" in this beam. From this observation we will extract from π two cycles of opposite effects by Pigeonhole Principle. Deleting these cycles shortens the length of τ. ◀

For thick runs, recall that they only occur in proper VASSes.

▶ **Lemma 4.26.** *Let G be a proper geometrically 2-dimensional d-VASS with* $\mathrm{supp}(\mathrm{Cyc}(G)) = [d]$*. For any $\mathbf{0}$-run τ in G that is A-thick for some $A \geq \chi(G)$, there is a $\mathbf{0}$-run ρ with the same source and target states as τ such that $|\rho| \leq A^{O(d^3)}$.*

Proof sketch. Consider a thick run τ. By definition, it can be decomposed into $\tau = \pi\pi'$ such that each of π and $\mathrm{rev}(\pi')$ contains four pumping cycles. For each path π and π', we remove any cycle whose effect can be realized by those pumping cycles to shorten their lengths. After that, we add some additional iterations to the pumping cycles to make the target configurations of π and π' meet in a common point. This is possible as we required that the sequential cones generated by the pumping cycles have a non-trivial intersection. The bound on the number of iterations of cycles can be obtained using standard tools in integer programming, e.g. [4] and [16]. ◀

With these bounds we can finish the proof of our major goal Lemma 4.9.

Proof of Lemma 4.9. By Lemma 4.14 and Lemma 4.19, any $\mathbf{0}$-run τ in G is either A-thin or A-thick for some $A = \chi(G)^{O(d\cdot\varsigma(G))}$. We can apply Lemma 4.25 or Lemma 4.26 to transform τ into a run ρ with the same source and target so that $|\rho| \leq A^{O(d^3)} \leq \chi(G)^{O(d^4\cdot\varsigma(G))}$. ◀

5 A Note on Geometrically 2-Dimensional 3-VASS

The projection techniques used in Section 4 did not provide a straightforward reduction from geometrically 2-dimensional d-VASS to 2-VASS. In this section we will show that such a reduction is indeed possible for $d = 3$. We shall also mention some issues one will face when trying to generalize this reduction for $d > 3$.

In this section we care about the unary-encoding size of VASSes. Let $G = (Q, T)$ be a d-VASS, its *unary-encoding size* is defined as $|G|_1 := |Q| + d \cdot |T| \cdot \|T\| + 1$. The reduction is stated as the following lemma, from which one immediately gets the PSPACE upper bound for reachability in geometrically 2-dimensinoal 3-VASS even under binary encoding.

▶ **Lemma 5.1.** *Given a geometrically 2-dimensinoal 3-VASS G with 2 states p, q, one can compute in time polynomial in $|G|_1$ a 2-VASS $\overline{G}$ with 2 states $\overline{p}, \overline{q}$ satisfying $|\overline{G}|_1 \leq |G|_1^{O(1)}$, such that the following statements are equivalent:*

- *there exists a run from $p(\mathbf{0})$ to $q(\mathbf{0})$ of length ℓ in G;*
- *there exists a run from $\overline{p}(\mathbf{0})$ to $\overline{q}(\mathbf{0})$ of length 3ℓ in $\overline{G}$.*

▶ **Corollary 5.2.** *Reachability in geometrically 2-dimensinoal 3-VASS is in* PSPACE.

Proof. By [2, Theorem 3.2], in a 2-VASS $\overline{G}$, length of the shortest reachability witness is bounded by $|\overline{G}|_1^{O(1)}$. Together with Lemma 5.1 this shows that length of the shortest reachability witness in a geometrically 2-dimensinoal 3-VASS G is also bounded by $|G|_1^{O(1)}$, which is exponential in the binary-encoding size $|G|$. Polynomial space is enough for enumerating every run of length at most $|G|_1^{O(1)}$. ◀

The following is devoted to Lemma 5.1. Fix a geometrically 2-dimensinoal 3-VASS $G = (Q, T)$ with 2 states p, q. We can assume that $\mathrm{gdim}(G) = 2$, by adding isolated loops to G when necessary. Then there exists a normal vector $\boldsymbol{n} \in \mathbb{Z}^3$ such that $\mathrm{Cyc}(G) = \{\boldsymbol{c} \in \mathbb{Q}^3 : \langle \boldsymbol{n}, \boldsymbol{c}\rangle = 0\}$. As $\mathrm{Cyc}(G)$ is spanned by effects of two simple cycles, whose norms are bounded by $\chi(G) < |G|_1^{O(1)}$, applying Cramer's rule we can assume $\|\boldsymbol{n}\| \leq |G|_1^{O(1)}$. Let $h \in \mathbb{Z}$, we define the set $C^h := \{\boldsymbol{u} \in \mathbb{N}^3 : \langle \boldsymbol{n}, \boldsymbol{u}\rangle = h\}$ as an affine copy of $\mathrm{Cyc}(G)$. The following proposition shows that any run from $p(\mathbf{0})$ moves within a limited number of C^h's.

▶ **Proposition 5.3.** *There exists a number $B \leq |G|_1^{O(1)}$ such that for any configuration $s(\boldsymbol{w})$ reachable from $p(\mathbf{0})$, $\boldsymbol{w} \in C^h$ for some h with $|h| \leq B$.*

Proof. By Lemma 3.4 we have $\boldsymbol{w} = \boldsymbol{c} + \boldsymbol{z}$ for some $\boldsymbol{c} \in \mathrm{Cyc}(G)$ and $\|\boldsymbol{z}\| \leq \chi(G)$. So $\boldsymbol{w} \in C^h$ for $h = \langle \boldsymbol{n}, \boldsymbol{w}\rangle$. Note that $|h| = |\langle \boldsymbol{n}, \boldsymbol{w}\rangle| = |\langle \boldsymbol{n}, \boldsymbol{c}\rangle + \langle \boldsymbol{n}, \boldsymbol{z}\rangle| = |\langle \boldsymbol{n}, \boldsymbol{z}\rangle|$. We can pick $B := 3\chi(G) \cdot \|\boldsymbol{n}\| \leq |G|_1^{O(1)}$. ◀

Proof of Lemma 5.1. Let B be the number given in Proposition 5.3. We consider 2 cases according to the signs of components in $\boldsymbol{n}$.

Case 1. $n \geq 0$ (or symmetrically $n \leq 0$). By Proposition 5.3, every configuration $s(\boldsymbol{w})$ reachable from $p(\mathbf{0})$ satisfies $|\langle \boldsymbol{n}, \boldsymbol{w}\rangle| \leq B$. We claim that $\left\|\boldsymbol{w}|_{\mathrm{supp}(\boldsymbol{n})}\right\| \leq B$. Otherwise, as $\boldsymbol{n}$ is an integer vector, $\boldsymbol{n}(i) \geq 1$ for all $i \in \mathrm{supp}(\boldsymbol{n})$, then $\left\|\boldsymbol{w}|_{\mathrm{supp}(\boldsymbol{n})}\right\| > B$ implies $|\langle \boldsymbol{n}, \boldsymbol{w}\rangle| > B$. Then we can encode all possible values of the coordinates in $\mathrm{supp}(\boldsymbol{n})$ using the states of the VASS and thus reduce its dimension. As $\boldsymbol{n} \neq \mathbf{0}$, $\mathrm{supp}(\boldsymbol{n})$ contains at least one coordinate, so the resulting VASS is at most 2-dimensional. And the size is blown up by a factor of at most $(B+1)^2 \leq |G|_1^{O(1)}$. Note that to validate the factor 3 in the amplification of lengths of runs as in the statements of Lemma 5.1, we also need to add two dummy transitions after each transition in the original VASS.

Case 2. n contains both positive and negative components. By negating $\boldsymbol{n}$ we can assume there is only one negative component in $\boldsymbol{n}$. Assume w.l.o.g. $\boldsymbol{n}(3) < 0$ and $\boldsymbol{n}(1), \boldsymbol{n}(2) \geq 0$. Let $\boldsymbol{n} =: (a, b, -c)$ for some $a, b, c \in \mathbb{N}$, then we have $C^h = \{(x, y, z) \in \mathbb{N}^3 : ax + by = cz + h\}$. We construct a 2-VASS $\overline{G}$ that projects G onto the first 2 coordinates. Let $\overline{G} = (\overline{Q}, \overline{T})$ be defined as follows. The state set $\overline{Q}$ contains $\{s^h : s \in Q, h \in \mathbb{Z}, |h| \leq B\} \cup \{\overline{s^h} : s \in Q, h \in \mathbb{Z}, |h| \leq B\}$. Intuitively a configuration $s^h(x, y)$ in $\overline{G}$ represents the configuration $s(x, y, z)$ in G such that $(x, y, z) \in C^h$, and the barred version marks an "unchecked" state, where $z = (ax + by - h)/c$ could be negative. So for each transition $t = (r, \boldsymbol{a}, s)$ of G, we add the transitions $t^h = (r^h, \boldsymbol{a}|_{\{1,2\}}, \overline{s^{h'}})$ to $\overline{G}$ for all $|h|, |h'| \leq B$ and $h' = h + \langle \boldsymbol{n}, \boldsymbol{a}\rangle$. Indeed, if $r(\boldsymbol{u}) \xrightarrow{t} s(\boldsymbol{v})$ in G and $\boldsymbol{u} \in C^h$, $\boldsymbol{v} \in C^{h'}$, then $h' = \langle \boldsymbol{n}, \boldsymbol{v}\rangle = \langle \boldsymbol{n}, \boldsymbol{u} + \Delta(t)\rangle = h + \langle \boldsymbol{n}, \Delta(t)\rangle$. This justifies the correctness of the transitions we added to $\overline{G}$.

Next, we add to $\overline{G}$ transitions that checks, at each state $\overline{s^h}$, if the configuration $\overline{s^h}(x, y)$ represents a legal configuration $s(x, y, z)$ in G, where $(x, y, z) \in C^h$, so $cz = ax + by - h$. As $c > 0$, this is equivalent to checking if $ax + by - h \geq 0$. Recall that $a, b \geq 0$, one can verify that for $(x, y) \geq \mathbf{0}$, we have $ax + by - h \geq 0$ if and only if $(x, y) \geq \boldsymbol{m}$ for some $\boldsymbol{m} \in \mathcal{M}^h$, where $\mathcal{M}^h$ is the set of minimal solutions given by

$$\mathcal{M}^h := \begin{cases} \{(0,0)\} & \text{if } h \leq 0; \\ \{(0, \lceil h/b\rceil)\} & \text{if } a = 0, b > 0, \text{ and } h > 0; \\ \{(\lceil h/a\rceil, 0)\} & \text{if } a > 0, b = 0, \text{ and } h > 0; \\ \{(x, \lceil (h - ax)/b\rceil) : x = 0, 1, \ldots, \lceil h/a\rceil\} & \text{if } a > 0, b > 0, \text{ and } h > 0. \end{cases} \tag{15}$$

Now for each number h with $|h| \leq B$, we add the path $\overline{s^h} \xrightarrow{-\boldsymbol{m}} \bullet \xrightarrow{\boldsymbol{m}} s^h$ to $\overline{G}$ for each $\boldsymbol{m} \in \mathcal{M}^h$, where $\bullet$ is an arbitrary but unique state added to $\overline{G}$. One can easily verify that $\overline{s^h}(x,y) \xrightarrow{*} s^h(x,y)$ if and only if $z = (ax+by-h)/c \geq 0$. Therefore, $p(\mathbf{0}) \xrightarrow{\pi} q(\mathbf{0})$ in G for a path π of length ℓ if and only if $p^0(\mathbf{0}) \xrightarrow{\overline{\pi}} q^0(\mathbf{0})$ in $\overline{G}$ for a path $\overline{\pi}$ of length 3ℓ. Also note that $|\mathcal{M}^h| \leq B \leq |G|_1^{O(1)}$, so we have $|\overline{G}|_1 \leq |G|_1^{O(1)}$. This completes the proof. ◀

Finally, we mention some issues when trying to generalize for higher dimensions.

1. If the dimension d is not fixed as a parameter, one can easily construct an acyclic (geometrically 0-dimensinoal) VASS so that each point in the Hamming cube $\{0,1\}^d$ is reachable from $\mathbf{0}$. Then the reduced VASS must handle an exponential number of affine copies of $\mathrm{Cyc}(G)$, which suggests an exponential blow-up in size.
2. For $d = 3$ the reduced 2-VASS only need to check inequalities of the form $\langle \boldsymbol{a}, \boldsymbol{x} \rangle \geq b$ for $\boldsymbol{a} \geq \mathbf{0}$. For higher dimensions we no longer have the assumption $\boldsymbol{a} \geq \mathbf{0}$. And we have no idea how to make a VASS check inequalities such as $x - y \geq 3$.

6 Geometrically 1-Dimensional and 0-Dimensional VASS

We have shown that the reachability problem in geometrically 2-dimensional VASS is PSPACE-complete, which is same as the complexity of reachability in 2-VASS. The situation becomes different for lower geometric dimensions. In this section we study the reachability problems for geometrically 1-dimensional and 0-dimensional VASSes. Indeed, these results turn out to be just rephrase of existing results for 2-VASS and 1-VASS.

▶ **Theorem 6.1.** *Reachability in geometrically 1-dimensional VASS is* PSPACE*-complete.*

Proof. The upper bound is implied by Theorem 4.2. For the lower bound, we refer the readers to the reduction from bounded one-counter automata to 2-VASS [3, Lemma 20]. We remark that as the reduced VASS uses 2 counters to simulate a bounded counter, every transition have effect of the form $(z, -z)$ for some $z \in \mathbb{Z}$. Thus, the cycle space of the reduced VASS is contained in $\mathrm{span}\{(1,-1)\}$. So this reduction indeed establishes the PSPACE-hardness of reachability in geometrically 1-dimensional VASS. ◀

▶ **Theorem 6.2.** *Reachability in geometrically 0-dimensional VASS is* NP*-complete.*

Proof. We first show the upper bound. Let $G = (Q, T)$ be a geometrically 0-dimensional VASS. Then any cycle in G has effect $\mathbf{0}$. So for any run τ in G we can safely remove all cycles from τ and obtain a run $\overline{\tau}$ whose length is bounded by $|Q|$. Now a nondeterministic algorithm can decide reachability in G by simply guessing a run of length at most $|Q|$.

For the lower bound, we recall the folklore reduction from Subset-Sum to the reachability problem in 1-VASS. Given an instance $\langle S = \{a_1, \ldots, a_n\}, s \rangle$ of Subset-Sum, where $S \subseteq \mathbb{N}$ and the goal is to find a subset of S whose sum is s, we construct a 1-VASS $G = (Q, T)$ as follows. The states are $Q := \{q_0, \ldots, q_n\}$. For each $i = 1, \ldots, n$ we add two transitions $q_{i-1} \xrightarrow{a_i} q_i$ and $q_{i-1} \xrightarrow{0} q_i$. One can easily observe that there exists a subset of S with sum s if and only if $q_0(0) \to^* q_n(s)$ in G. Note that there are no cycles in G, thus G is indeed geometrically 0-dimensional. ◀

7 Concluding Remarks

In this paper, we have studied the reachability problem in vector addition systems with states (VASS) parameterized by geometric dimension. We introduced an efficient algorithm for computing the geometric dimension of a VASS and demonstrated some simple geometrical

properties of reachable sets and runs. The primary focus is on VASS with low geometric dimensions, particularly those that are geometrically 2-dimensional. By generalizing existing pumping techniques for 2-VASS, we have shown that the reachability problem in geometrically 2-dimensional VASS is PSPACE-complete. The techniques of sign-reflecting projection and support projection also provide insights into how results for d-VASS can be adapted to geometrically d-dimensional VASS.

The results for geometrically 1- and 0-dimensional VASS are both interesting and technically straightforward. By re-examining existing results, we have shown that reachability in geometrically 1-dimensional VASS is PSPACE-complete, while in geometrically 0-dimensional VASS it is NP-complete. It is worth noting that reachability is known to be NP-complete in 1-VASS and NL-complete in 0-VASS. Our findings highlight a distinction in expressiveness and computational power between geometrically d-dimensional VASS and d-VASS. We suggest that comparing these two models could be a possible direction for future research.

In this paper our main focus is on binary encoded VASSes of low geometric dimensions. The situation would become different if unary encoding is considered. It is known that in fixed-dimensional settings the problem is NL-complete for dimensions 0, 1, and 2 [2]. However, if only the geometric dimension is fixed, we note that logarithmic space is not enough to store a single configuration containing d counters (recall that the size of $G = (Q, T)$ under unary encoding is $|G|_1 := |Q| + d \cdot |T| \cdot \|T\| + 1$). The complexity of reachability in low geometric dimensions under unary encoding is left for future work.

References

1 Shaull Almagor, Arka Ghosh, Tim Leys, and Guillermo A. Pérez. The geometry of reachability in continuous vector addition systems with states. *Inf. Comput.*, 304:105298, 2025. doi:10.1016/J.IC.2025.105298.

2 Michael Blondin, Matthias Englert, Alain Finkel, Stefan Göller, Christoph Haase, Ranko Lazic, Pierre McKenzie, and Patrick Totzke. The reachability problem for two-dimensional vector addition systems with states. *J. ACM*, 68(5):34:1–34:43, 2021. doi:10.1145/3464794.

3 Michael Blondin, Alain Finkel, Stefan Göller, Christoph Haase, and Pierre McKenzie. Reachability in two-dimensional vector addition systems with states is pspace-complete. In *30th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2015, Kyoto, Japan, July 6-10, 2015*, pages 32–43. IEEE Computer Society, 2015. doi:10.1109/LICS.2015.14.

4 Dmitry Chistikov and Christoph Haase. The taming of the semi-linear set. In Ioannis Chatzigiannakis, Michael Mitzenmacher, Yuval Rabani, and Davide Sangiorgi, editors, *43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016, July 11-15, 2016, Rome, Italy*, volume 55 of *LIPIcs*, pages 128:1–128:13. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPICS.ICALP.2016.128.

5 Wojciech Czerwinski, Ismaël Jecker, Slawomir Lasota, Jérôme Leroux, and Lukasz Orlikowski. New lower bounds for reachability in vector addition systems. In Patricia Bouyer and Srikanth Srinivasan, editors, *43rd IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2023, December 18-20, 2023, IIIT Hyderabad, Telangana, India*, volume 284 of *LIPIcs*, pages 35:1–35:22. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICS.FSTTCS.2023.35.

6 Wojciech Czerwinski, Slawomir Lasota, Christof Löding, and Radoslaw Piórkowski. New pumping technique for 2-dimensional VASS. In Peter Rossmanith, Pinar Heggernes, and Joost-Pieter Katoen, editors, *44th International Symposium on Mathematical Foundations of Computer Science, MFCS 2019, August 26-30, 2019, Aachen, Germany*, volume 138 of *LIPIcs*, pages 62:1–62:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPICS.MFCS.2019.62.

7 Wojciech Czerwinski and Lukasz Orlikowski. Reachability in vector addition systems is ackermann-complete. In *62nd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2021, Denver, CO, USA, February 7-10, 2022*, pages 1229–1240. IEEE, 2021. doi:10.1109/FOCS52979.2021.00120.

8 Wojciech Czerwiński, Ismaël Jecker, Sławomir Lasota, and Łukasz Orlikowski. Reachability in 3-vass is elementary. *CoRR*, abs/2502.13916, 2025. doi:10.48550/arXiv.2502.13916.

9 Matthias Englert, Ranko Lazic, and Patrick Totzke. Reachability in two-dimensional unary vector addition systems with states is nl-complete. In Martin Grohe, Eric Koskinen, and Natarajan Shankar, editors, *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016*, pages 477–484. ACM, 2016. doi:10.1145/2933575.2933577.

10 Javier Esparza and Mogens Nielsen. Decidability issues for petri nets - a survey. *J. Inf. Process. Cybern.*, 30(3):143–160, 1994.

11 John Fearnley and Marcin Jurdzinski. Reachability in two-clock timed automata is pspace-complete. *Inf. Comput.*, 243:26–36, 2015. doi:10.1016/J.IC.2014.12.004.

12 Yuxi Fu, Qizhe Yang, and Yangluo Zheng. Improved algorithm for reachability in d-vass. In Karl Bringmann, Martin Grohe, Gabriele Puppis, and Ola Svensson, editors, *51st International Colloquium on Automata, Languages, and Programming, ICALP 2024, July 8-12, 2024, Tallinn, Estonia*, volume 297 of *LIPIcs*, pages 136:1–136:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPICS.ICALP.2024.136.

13 Christoph Haase, Stephan Kreutzer, Joël Ouaknine, and James Worrell. Reachability in succinct and parametric one-counter automata. In Mario Bravetti and Gianluigi Zavattaro, editors, *CONCUR 2009 - Concurrency Theory, 20th International Conference, CONCUR 2009, Bologna, Italy, September 1-4, 2009. Proceedings*, volume 5710 of *Lecture Notes in Computer Science*, pages 369–383. Springer, 2009. doi:10.1007/978-3-642-04081-8_25.

14 Jérôme Leroux. The reachability problem for petri nets is not primitive recursive. In *62nd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2021, Denver, CO, USA, February 7-10, 2022*, pages 1241–1252. IEEE, 2021. doi:10.1109/FOCS52979.2021.00121.

15 Jérôme Leroux and Sylvain Schmitz. Reachability in vector addition systems is primitive-recursive in fixed dimension. In *34th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2019, Vancouver, BC, Canada, June 24-27, 2019*, pages 1–13. IEEE, 2019. doi:10.1109/LICS.2019.8785796.

16 Loic Pottier. Minimal solutions of linear diophantine systems: Bounds and algorithms. In Ronald V. Book, editor, *Rewriting Techniques and Applications, 4th International Conference, RTA-91, Como, Italy, April 10-12, 1991, Proceedings*, volume 488 of *Lecture Notes in Computer Science*, pages 162–173. Springer, 1991. doi:10.1007/3-540-53904-2_94.

17 Sylvain Schmitz. Complexity hierarchies beyond elementary. *CoRR*, abs/1312.5686, 2013. arXiv:1312.5686.

18 Qizhe Yang and Yuxi Fu. Reachability in 3-vass is in tower. *CoRR*, abs/2306.05710, 2023. doi:10.48550/arXiv.2306.05710.

19 Richard Zurawski and MengChu Zhou. Petri nets and industrial applications: A tutorial. *IEEE Trans. Ind. Electron.*, 41(6):567–583, 1994. doi:10.1109/41.334574.

Denotational Semantics for Probabilistic and Concurrent Programs

Noam Zilberstein
Cornell University, Ithaca, NY, USA

Daniele Gorla
Sapienza Università di Roma, Italy

Alexandra Silva
Cornell University, Ithaca, NY, USA
University College London, UK

Abstract

We develop a denotational model for *probabilistic and concurrent imperative programs*, a class of programs with standard control flow via conditionals and while-loops, as well as probabilistic actions and parallel composition. Whereas semantics for concurrent or randomized programs in isolation is well studied, their combination has not been thoroughly explored and presents unique challenges. The crux of the problem is that interactions between control flow, probabilistic actions, and concurrent execution cannot be captured by straightforward generalizations of prior work on pomsets and convex languages, prominent models for those effects, individually. Our model has good domain theoretic properties, important for semantics of unbounded loops. We also prove two adequacy theorems, showing that the model subsumes typical powerdomain semantics for concurrency and convex powerdomain semantics for probabilistic nondeterminism.

2012 ACM Subject Classification Theory of computation → Parallel computing models; Theory of computation → Denotational semantics; Theory of computation → Probabilistic computation

Keywords and phrases Denotational Semantics, Pomsets, Concurrency, Convex Powerset

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2025.39

Related Version *Full Version*: `https://doi.org/10.48550/arXiv.2503.02768` [67]

Funding This work was partially supported ARIA's Safeguarded AI programme.

1 Introduction

From simple imperative languages, to functional ones, to probabilistic and quantum paradigms, the development of programming languages is often accompanied by a study of mathematical objects capturing the semantics of syntactic constructs. Program semantics underpins many static analysis and verification techniques, and also how a language can be extended.

Semantics can be presented in different styles – denotational, operational, and axiomatic – and each approach offers different insights. Denotational semantics is often chosen for expressivity and extensibility. For example, adding recursion raises questions about the underlying *domain*, motivating the development of domain theory to provide mathematical representations of iterated computations [51, 52]. Other constructs necessitate more involved domains; two prime examples are probabilistic and concurrent programs.

Probabilistic programs have been widely studied – going back to the seminal works of Kozen [31] and McIver and Morgan [36] – and their semantic domains require extra algebraic structure to capture the *distribution* of outcomes generated by operations such as sampling and coin flips. This added convex structure brings additional complexity to questions of language extensions and program analysis, but the last decade has brought success stories on taming the complexity through program logics and predicate transformer calculi [5, 10, 36, 41, 68].

© Noam Zilberstein, Daniele Gorla, and Alexandra Silva;
licensed under Creative Commons License CC-BY 4.0

Concurrent programs have prominent applications in distributed systems and have long been a target of program analysis, as concurrency bugs are hard to detect and prevent. There have been many approaches to concurrent semantics, some arising from process algebra research and some from more practical hardware considerations (*e.g.,* memory models). In terms of denotational semantics, *pomsets* [8, 17, 18, 47] provide a prominent model, expressing causality between actions and parallel branching as a partial order.

Semantics of concurrent and probabilistic programs, separately, are subtle and complex. Unsurprisingly, their combination introduces additional challenges. In this paper, we develop a denotational model for programs that mix probabilistic and concurrent constructs. We work with a simple concurrent imperative programming language, shown below.

$$\mathit{cmd} \ni C ::= \textbf{skip} \mid C_1; C_2 \mid \ C_1 \,|\, C_2 \ \mid \textbf{if } b \textbf{ then } C_1 \textbf{ else } C_2 \mid \textbf{while } b \textbf{ do } C \mid a \tag{1}$$

Here, the basic actions $a \in \mathit{act}$ can perform probabilistic operations such as random sampling. This is not only interesting from a theoretical point of view; for decades, randomization has been used to enhance the capabilities of concurrent programs [19, 43, 49, 50]. For example, Dijkstra's famous *Dining Philosophers Problem* – a distributed synchronization scheme – has no deterministic solution [11], but it has a simple randomized one [35].

Despite the importance of mixing randomization and concurrency, semantic models for probabilistic concurrent programs with unbounded looping are not yet well understood. Such models are necessary to analyze behaviors of many concurrent randomized protocols – including the Dining Philosophers – which avoid deadlock with probability 1, but admit an infinite trace whose probability converges to 0. In this paper, we develop such a model, drawing insights from prior work on concurrent and probabilistic semantics, but solving challenges unique to their combination. In a nutshell, the contributions of this paper are:

1. We introduce *pomsets with formulae*, and prove that they have well behaved domain-theoretic properties (Section 3), including an *extension lemma* [37], for extending monotone operations on finite pomsets to continuous operations on infinite ones.
2. We define guarded, sequential, and parallel composition operations on pomsets with formulae (Section 4) and use those operations to define the denotational semantics for the concurrent imperative language (1), with uninterpreted actions (Section 5.1).
3. We define a linearization operation (via the extension lemma), parametric on the computational domain, to convert the pomset semantics into a state transformer (Section 5.2).
4. We prove two adequacy theorems showing that our model captures established models for probabilistic nondeterminism (Theorem 5.3) and pure concurrency (Theorem 6.1).

We discuss related work in Section 7; omitted proofs and details are provided in [67].

2 An Overview of Probabilistic Concurrency Semantics

In this section, we introduce the requirements for interpreting programs that are both probabilistic and concurrent. To illustrate the subtleties, consider the following program:

$$C \triangleq x :\approx \textbf{flip}\left(\tfrac{1}{2}\right); (y := 0 \,|\, y := 1); \textbf{if } y \textbf{ then skip else } x := x + 2 \tag{2}$$

After executing C, the value of x can be any integer between 0 and 3, and that value depends both on random sampling and nondeterminism (from concurrent scheduling). The parity of x is fixed after the coin flip (returning 0 or 1 with probability $\frac{1}{2}$), but then the scheduler can influence its final value by choosing in which order to run the updates to y. Importantly though, x will be even with probability exactly $\frac{1}{2}$, regardless of the behavior of the scheduler.

Any semantic domain for this program must encompass probability distributions of program states, while at the same time accounting for nondeterminism introduced from parallel branching. One might attempt to give semantics to this program using a naive composition of distributions and powerset (as monads) but, as it is well known from prior work, such combination can easily lead to non-compositional semantics [45, 63, 71, 72]. While there are several domains for this combination of effects, we use the *convex powerset*, which is quite well-studied in that it is a monad [23] for sequential composition; it is a directed complete partial order (DCPO) [30, 57–59] for finding fixed points of iterated computations; and has well behaved equational laws [6, 39, 40].

We now give a basic account of the convex powerset; for more details on the DCPO structure and monad operations, refer to Section A.2 and [68, §3]. A discrete probability distribution $\mu \in \mathcal{D}(X) = X \to [0,1]$ is a countable map from X to probabilities such that $\sum_{x \in X} \mu(x) = 1$. Convex combinations are defined for distributions $(\mu \oplus_p \nu)(x) = p \cdot \mu(x) + (1-p) \cdot \nu(x)$ and sets of distributions $S \oplus_p T \triangleq \{\mu \oplus_p \nu \mid \mu \in S, \nu \in T\}$. A set $S \subseteq \mathcal{D}(X)$ is *convex* if $S = S \oplus_p S$ for all $p \in [0,1]$, *i.e.,* S contains all convex combinations of its elements. Our computational domain – *the convex powerset* – consists of nonempty convex sets of distributions[1].

$$\mathcal{C}(X) \triangleq \{S \subseteq \mathcal{D}(X \cup \{\bot\}) \mid S \text{ is nonempty, convex, } \dots\}$$

This computational domain allows us give semantics to the coin flip actions in program (2) $[\![a]\!]_{act} : \mathcal{S} \to \mathcal{C}(\mathcal{S})$ where $\mathcal{S} \triangleq \mathit{var} \to \mathit{val}$ is the set of variable valuations. The semantics is given by a set containing a single distribution, where x is set to 1 with probability p and it is set to 0 with probability $1-p$. Being a *set of distributions*, $\mathcal{C}(\mathcal{S})$ can also represent nondeterministic choice, which we denote by $\&$. Operationally, $S \,\&\, T$ is not simply a choice between S and T, but rather it is a choice of a *distribution* over S and T, corresponding to a scheduler that can use biased coin flips to make decisions [60].

$$[\![x :\approx \mathbf{flip}\,(p)]\!]_{act}\,(s) \triangleq \left\{ \left[\begin{array}{lcl} s[x \coloneqq 1] & \mapsto & p \\ s[x \coloneqq 0] & \mapsto & 1-p \end{array} \right] \right\} \qquad S \,\&\, T \triangleq \bigcup_{p \in [0,1]} S \oplus_p T$$

Our goal is to develop a pomset model to compositionally reason about concurrency in a manner that is compatible with the convex powerset. Existing pomset models do not track control flow, and instead rely on the following equation, stating that a command followed by an if-statement can be decomposed into two traces, with the split lifted to the top:

$$C; \mathbf{if}\ b\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2 \quad \equiv \quad (C; \mathbf{assume}\ b; C_1) \,\&\, (C; \mathbf{assume}\ \neg b; C_2) \tag{3}$$

It is well known that this equation is invalid in probabilistic contexts, especially $\mathcal{C}$ [39]. More concretely, prior denotational models for concurrency $[\![-]\!]_{\mathsf{PL}} : \mathit{cmd} \to \mathcal{P}(\mathit{pom})$ (shown in Figure 5) map programs to *pomset languages* – sets of pomsets – where each $\boldsymbol{\alpha} \in [\![C]\!]_{\mathsf{PL}}$ corresponds to a particular resolution of the tests in the program. For program (2), this approach yields the semantics on the left of Figure 1, where the problematic Equation (3) is implicitly applied. Arrows $a_1 \to a_2$ indicate causality, *i.e.,* a_1 must occur before a_2; when there is no arrow between two actions, they can be interleaved in any order. There is a significant problem in $[\![C]\!]_{\mathsf{PL}}$; as we will soon see, there is no straightforward way to piece together the entire distribution of outcomes in the program.

[1] The DCPO structure relies of a few more properties, which we report in Section A.2.

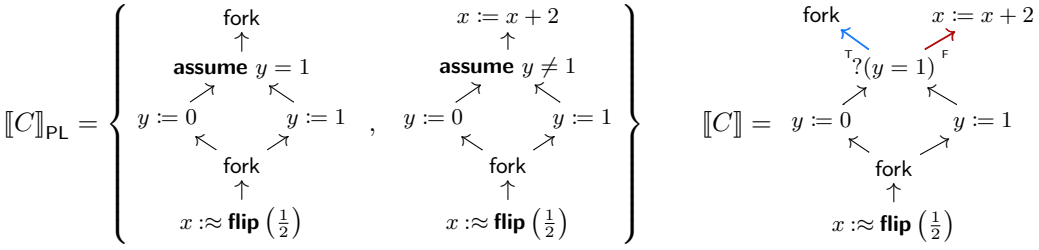


Figure 1 Pomset language model (left) vs pomset with formulae model (right) for program (2) (we use a fork with 0 branches to represent **skip**).

Crucially, our pomset structure tracks the results of control-flow tests (which can rely on randomization). This is done symbolically by associating a Boolean formula to each node of the pomset, recording the resolution of tests needed to reach that point. We call these new structures *pomsets with formulae*, which capture both parallel composition and guarded branching[2]. As such, this structure encodes many traces, avoiding the need for a set of traces and simultaneously remaining compositional in the presence of probabilistic actions.

The semantics in our new model $[\![-]\!] : \mathit{cmd} \to \mathit{pom}$ for program (2) is shown on the right of Figure 1. Compared to $[\![C]\!]_{\mathsf{PL}}$, our new model has merged the set of structures into a single structure, where the two opposing tests are replaced by a single node labelled $?(y = 1)$. The outgoing edges are labelled T and F, indicating that they can only be followed if the test passes or fails, respectively. Now, we may wish to know the possible probabilities for each outcome resulting from running this program. To this end, in Section 5.2 we develop a *linearization* procedure $\mathcal{L} \colon \mathit{pom} \to \mathcal{S} \to \mathcal{C}(\mathcal{S})$ for interpreting a pomset as a state transformer. Linearizing this structure, we get the following set of distributions:

$$\mathcal{L}([\![C]\!])(s) = \left\{ \begin{bmatrix} s[x := 0, y := 0] & \mapsto p \\ s[x := 1, y := 0] & \mapsto q \\ s[x := 2, y := 1] & \mapsto \frac{1}{2} - p \\ s[x := 3, y := 1] & \mapsto \frac{1}{2} - q \end{bmatrix} \middle| \; 0 \leq p, q \leq \frac{1}{2} \right\}$$

Linearization provides new insights about the program, which were not obvious in the pomset structure. The scheduler can make y equal to 0 or 1 with any probability, which matches our operational understanding of the program: the scheduler can choose to execute the commands in either the order $y := 0; y := 1$ or $y := 1; y := 0$ (or some convex combination thereof). However, regardless of the scheduler's choice of p and q, x is even (or odd) with exactly probability $\frac{1}{2}$. This formal semantics matches the intuition for the program, since the parity of x is fixed by the initial coin flip.

We now contrast the semantics above with pomset language semantics to show that the program C cannot be meaningfully interpreted using prior techniques. Recall that in $[\![C]\!]_{\mathsf{PL}}$ each pomset corresponds to a particular resolution of the test, *i.e.,* $y = 1$ and $y \neq 1$. So, in the first pomset we will always have $x \leq 1$ whereas in the second pomset $x \geq 2$. Let us now try to obtain a meaningful state transformer semantics from this program by linearizing each structure and then merging the results. To do so, we use the following semantics for **assume**,

[2] Formulae are more expressive than conflict relations, *e.g.,* in event structures: conflict can only exclude execution of branches, whereas formulae record the entire outcome of tests that lead to every node.

where failed tests evaluate to a special **?** symbol, which halts the program execution:

$$[\![\textbf{assume } b]\!]_{act}(s) = \begin{cases} \eta(s) & \text{if } [\![b]\!]_{test}(s) = 1 \\ \eta(\textbf{?}) & \text{if } [\![b]\!]_{test}(s) = 0 \end{cases}$$

Now, letting $\boldsymbol{\alpha}_1$ and $\boldsymbol{\alpha}_2$ be the left and right pomsets in $[\![C]\!]_{\mathsf{PL}}$, we apply linearization to both structures to obtain:

$$\mathcal{L}(\boldsymbol{\alpha}_1)(s) = \left\{ \left[\begin{array}{ccc} s[x \coloneqq 0, y \coloneqq 0] & \mapsto & p_1 \\ s[x \coloneqq 1, y \coloneqq 0] & \mapsto & q_1 \\ \textbf{?} & \mapsto & 1 - p_1 - q_1 \end{array} \right] \middle| \; 0 \le p_1, q_1 \le \frac{1}{2} \right\}$$

$$\mathcal{L}(\boldsymbol{\alpha}_2)(s) = \left\{ \left[\begin{array}{ccc} s[x \coloneqq 2, y \coloneqq 1] & \mapsto & p_2 \\ s[x \coloneqq 3, y \coloneqq 1] & \mapsto & q_2 \\ \textbf{?} & \mapsto & 1 - p_2 - q_2 \end{array} \right] \middle| \; 0 \le p_2, q_2 \le \frac{1}{2} \right\}$$

Already, we can begin to see a problem; in $\mathcal{L}([\![C]\!])(s)$ the scheduler picks two probabilities, but here the scheduler picks four probabilities, giving it a higher degree of freedom to influence the outcome of the program. We make this formal by attempting to define an operation $\bowtie$ to merge the two semantics. As a first attempt, we generate a new set of distributions by summing the non-**?** probability mass in every pair of distributions from the two sets such that the mass of **?** in one is equal to the non-**?** mass in the other:

$$S \bowtie T \triangleq \{\mu \bowtie \nu \mid \mu \in S, \nu \in T, \mu(\textbf{?}) = 1 - \nu(\textbf{?})\} \quad (\mu \bowtie \nu)(s) \triangleq \begin{cases} \mu(s) + \nu(s) & \text{if } s \neq \textbf{?} \\ 0 & \text{if } s = \textbf{?} \end{cases}$$

Indeed, this gives us $\mathcal{L}([\![C]\!])(s) \subseteq \mathcal{L}(\boldsymbol{\alpha}_1)(s) \bowtie \mathcal{L}(\boldsymbol{\alpha}_2)(s)$: take any $\mu \in \mathcal{L}([\![C]\!])(s)$, which is generated by fixing some probabilities $0 \le p, q \le \frac{1}{2}$. Picking $p_1 = p$, $q_1 = q$, $p_2 = \frac{1}{2} - p$, and $q_2 = \frac{1}{2} - q$, we see that $\mu \in \mathcal{L}(\boldsymbol{\alpha}_1)(s) \bowtie \mathcal{L}(\boldsymbol{\alpha}_2)(s)$. However, $\mathcal{L}(\boldsymbol{\alpha}_1)(s) \bowtie \mathcal{L}(\boldsymbol{\alpha}_2)(s)$ contains extra distributions that are not in $\mathcal{L}([\![C]\!])(s)$, and are not correct outcomes of the program. For example, letting $p_1 = p_2 = \frac{1}{2}$ and $q_1 = q_2 = 0$, we get:

$$\left[\begin{array}{ccc} s[x \coloneqq 0, y \coloneqq 0] & \mapsto & \frac{1}{2} \\ s[x \coloneqq 1, y \coloneqq 0] & \mapsto & 0 \\ s[x \coloneqq 2, y \coloneqq 1] & \mapsto & \frac{1}{2} \\ s[x \coloneqq 3, y \coloneqq 1] & \mapsto & 0 \end{array} \right] \in \mathcal{L}(\boldsymbol{\alpha}_1)(s) \bowtie \mathcal{L}(\boldsymbol{\alpha}_2)(s)$$

But this distribution cannot be an outcome of running the program, since x is even with probability 1, deviating from the expected operational behavior!

So a different implementation of $\bowtie$ is needed, which is more discerning about the compatibility of pairs of distributions from the two sets. But as it stands, there is no information in $\mathcal{L}(\boldsymbol{\alpha}_1)(s)$ and $\mathcal{L}(\boldsymbol{\alpha}_2)(s)$ to indicate compatibility. While we do not claim that it is impossible to record this information during linearization, our investigation suggests that it would not be straightforward. A proper linearization of the pomset language $[\![C]\!]_{\mathsf{PL}}$ would essentially amount to a lockstep execution of the two pomsets, so that both branches can be taken with the appropriate probabilities. That lockstep execution corresponds to a straightforward linearization of the single pomset with formulae $[\![C]\!]$, where each test already has information about both branches. We therefore conclude that pomsets with formulae are the correct semantic structure for this domain. In the remainder of the paper, we develop the formal details of our pomset with formulae model and associated linearization procedure.

3 Semantic Structures

In this section, we define a particular kind of labelled partial order, which will form the core of our semantic model. We call this structure a *labeled partial order with formulae* (LPOF), as it includes a special labelling, assigning a Boolean formula to each node. We recall basic definitions from domain theory in Section A.1; for a complete treatment refer to [1].

3.1 Labelled Partial Orders with Formulae

Let *nodes* be a countable universe of nodes, that will be denoted by $x, y, z, \ldots$, whereas subsets of *nodes* will be denoted by $N, X, Y, \ldots$. We start by defining the class of Boolean formulae using nodes as the variables:

$$\mathit{form} \ni \psi ::= \mathsf{true} \mid \mathsf{false} \mid \psi_1 \wedge \psi_2 \mid \psi_1 \vee \psi_2 \mid \neg\psi \mid x$$

Formulae are interpreted by valuations $v \colon \mathit{nodes} \to \mathbb{B}$, where $\mathbb{B} = \{0, 1\}$; the satisfaction relation, written $v \vDash \psi$, is defined in the standard way (see Definition A.5 in Section A.1). The variables of a formula ψ, written $\mathsf{vars}(\psi)$, are those nodes that appear in ψ. We write $\mathsf{sat}(\psi)$ iff there exists a valuation v such that $v \vDash \psi$; $\psi \Rightarrow \psi'$ iff $v \vDash \psi'$, for every $v \vDash \psi$; and $\psi \Leftrightarrow \psi'$ iff $\psi \Rightarrow \psi'$ and $\psi' \Rightarrow \psi$. In what follows, given a strict poset $\langle X, < \rangle$ and any $x \in X$, we let $x{\downarrow} \triangleq \{y \in X \mid y < x\}$, $x{\uparrow} \triangleq \{y \in X \mid x < y\}$, $\mathsf{succ}(x) \triangleq \{y \in X \mid x < y \wedge \nexists z.(x < z < y)\}$, and $\min \triangleq \{x \in X \mid x{\downarrow} = \emptyset\}$. Further, the level of x, written $\mathsf{lev}(x)$, is the length of the longest path from a minimal element to x. See Section A.1 for more details.

▶ **Definition 3.1** (LPOF). *Let $\langle L, \leq \rangle$ be a pointed, finitely preceded DCPO with bottom element $\bot$. A labelled partial order with formulae (LPOF) over L is a 4-tuple $\alpha = \langle N, <, \lambda, \varphi \rangle$ where:*

1. *$N \subseteq \mathit{nodes}$ is a countable set of nodes;*
2. *$\langle N, < \rangle$ is a (strict) poset such that:*
   a. *it is finitely preceded, that is $|x{\downarrow}| < \infty$, for every $x \in N$;*
   b. *every level has a finite number of nodes,* i.e., *$|\mathsf{lev}^{-1}(n)| < \infty$ for all $n \in \mathbb{N}$; and*
   c. *it is single-rooted, that is: $|\min| = 1$.*
3. *$\lambda \colon N \to L$ is a labelling function such that $\mathsf{succ}(x) = \emptyset$ whenever $\lambda(x) = \bot$.*
4. *$\varphi \colon N \to \mathit{form}$ is a formula function satisfying:*
   a. *$\mathsf{sat}(\varphi(x))$ and $\mathsf{vars}(\varphi(x)) \subseteq x{\downarrow}$, for all $x \in N$;*
   b. *$\varphi(y) \Rightarrow \varphi(x)$, for all $x < y$.*

We denote by $\mathit{lpof}(L)$ the set of all LPOFs over L.

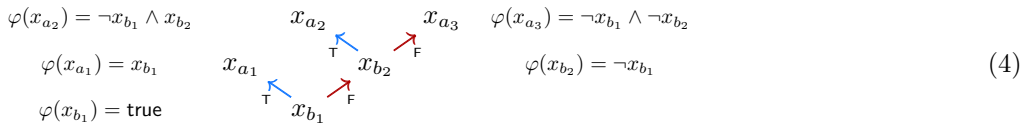
For some LPOF $\alpha = \langle N, <, \lambda, \varphi \rangle$, we use N_α, $<_\alpha$, λ_α, and φ_α to refer to its constituent parts; similarly, we annotate all functions with the LPOF they refer to, *e.g.,* $x{\downarrow_\alpha}$, $\mathsf{succ}_\alpha(x)$, $\min_\alpha$, etc. We now explain the conditions in Definition 3.1.

As is typical in pomset semantics, the partial order $\langle N, < \rangle$ denotes *causality*: $x < y$ iff x must be scheduled before y. If $x \not< y$ and $y \not< x$, then x and y execute concurrently, and can be interleaved in any order. With this in mind, conditions (2a) and (2b) are standard; in particular, since a node represents an action of a program, having finitely many predecessors ensures that every action may happen in a finite amount of time. Moreover, requiring that every level has a finite number of nodes is a weakening of the usual finite branching property: indeed, we allow a node to have infinitely many successors, but they cannot be all at the same level. We defer the discussion on Condition (2c) to Remark 4.1 later on.

Concerning Condition (3), labels correspond to *Boolean tests* and *actions*, performed during a program execution (*e.g.,* Figure 1). Unlike standard pomset models, actions can be probabilistic, and our model is consistent with known sequential probabilistic semantics

(Theorem 5.3). For the moment, we leave labels unspecified, only assuming the presence of $\bot$, used to denote a nonterminating computation, which is fundamental in approximating the fixed point semantics of while-loops. Since $\bot$ denotes nontermination, nodes labelled with $\bot$ cannot be a predecessor of any node, as the successors of $\bot$ would never be executed.

Condition (4) is about formulae, which are crucial for modeling conditional constructs (if-then-else and while-loops), such as **if** b_1 **then** a_1 **else if** b_2 **then** a_2 **else** a_3, where b_1 and b_2 are tests and a_1, a_2, a_3 are actions. This program corresponds to the LPOF below, where $\lambda(x_\ell) = \ell$ for all $\ell \in \{b_1, b_2, a_1, a_2, a_3\}$, $w \xrightarrow{\mathsf{T}} w_1$ means that $\varphi(w_1) \Leftrightarrow \varphi(w) \wedge w$ and $w \xrightarrow{\mathsf{F}} w_2$ means that $\varphi(w_2) \Leftrightarrow \varphi(w) \wedge \neg w$ (*i.e.,* w is labeled with a test, whose outcome leads to w_1 or w_2, depending on the Boolean outcome depicted on the colored arc):



(4)

Hence, every node x labelled with a Boolean test yields a (binary) branch and its (two) successors have the same formula as x, with an extra conjunct that is either x or $\neg x$, according to the outcome of the test. With this in mind, Definition 3.1(4) is quite intuitive:

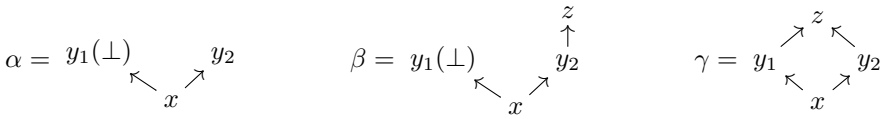
- Requiring $\mathsf{sat}(\varphi(x))$ amounts to expressing the fact that every node x is reachable, *i.e.,* there exists a truth assignment v such that $v \models \varphi(x)$. The truth assignment tells, for every node labeled with a Boolean test, whether that test passes or not. In our example, x_{a_3} is reachable when both the tests b_1 and b_2 fail; this is exactly what satisfiability of $\varphi(x_{a_3})$ entails (viz., that both b_1 and b_2 must be false to satisfy $\neg x_{b_1} \wedge \neg x_{b_2}$).
- This intuition justifies the other two requirements of Definition 3.1(4): reachability of a node can only depend on the values of the tests that precede it (and so $\varphi(x)$ can only use nodes that precede x) and, since along a path we shall only add conjuncts, the formula of a higher-level node is stronger than the formulae of all of its predecessors.

Next, we define an order on LPOFs, which we will use for the construction of fixed points to give semantics to unbounded loops. The intuition is that $\alpha \sqsubseteq_{lpof} \beta$ iff β has *more behaviors* than α, meaning that β can be obtained by expanding $\bot$ nodes in α into larger structures (each unfolding of a while-loop is obtained in this way, as shown in Section 5.1). Since LPOFs can only be expanded from $\bot$ nodes, the label set L must be pointed (*i.e.,* $\bot \in L$); however, the order on L need not be flat: labels themselves can also become larger when passing from α to β (*e.g.,* if actions are nondeterministic assignments from a given set, this corresponds to enlarging the set of possible choices). This can also be useful to model *invariant sensitive execution*, introduced in Probabilistic Concurrent Outcome Logic [70]. Notationally, Bot_α denotes the set of nodes labelled with $\bot$, *i.e.,* $\mathsf{Bot}_\alpha \triangleq \{x \in N_\alpha \mid \lambda_\alpha(x) = \bot\}$.

▶ **Definition 3.2** (Ordering on LPOFs). *We let* $\alpha \sqsubseteq_{lpof} \beta$ *iff:*

1. N_α *is a downward-closed subset of* N_β*, written* $N_\alpha \subseteq_\downarrow N_\beta$*;*
2. $<_\alpha \; = \; <_\beta \cap (N_\alpha \times N_\alpha)$*;*
3. $\forall x \in N_\alpha$*:*
   a. $\lambda_\alpha(x) \leq \lambda_\beta(x)$*;*
   b. $\varphi_\alpha(x) = \varphi_\beta(x)$*;*
   c. $\mathsf{succ}_\alpha(x) = \mathsf{succ}_\beta(x) \setminus \mathsf{Bot}_\alpha{\uparrow_\beta}$.

▶ **Example 3.3.** Consider the following three LPOFs:



where node y_1 is labelled with $\bot$ in α and β, whereas all other nodes have non-$\bot$ labels. Following the intuition given above, both α and β are attempts to specify that the computation in γ is truncated at y_1. However, only $\alpha \sqsubseteq_{\mathit{lpof}} \gamma$; this is because, if $\bot$ represents a diverging computation, then z will never have all the causes it needs to be executed and so it must disappear. Indeed, $\beta \not\sqsubseteq_{\mathit{lpof}} \gamma$ because, even though $N_\beta \subseteq_\downarrow N_\gamma$, condition (3c) of Definition 3.2 is violated: $z \in \mathsf{succ}_\beta(y_2)$ but $z \notin \mathsf{succ}_\gamma(y_2) \setminus \mathsf{Bot}_\beta{\uparrow_\gamma}$, since $y_1 \in \mathsf{Bot}_\beta$ and $y_1 <_\gamma z$.

Finally, in order to find fixed points, we will need to have a directed complete partial order (DCPO) structure and a notion of continuity for functions, which we now define.

▶ **Definition 3.4** (DCPO)**.** *Given a poset $\langle X, \leq\rangle$, a subset $D \subseteq X$ is called* directed *iff it is not empty and, for every two elements $x_1, x_2 \in D$, there exists $x \in D$ such that $x_1, x_2 \leq x$.*

$\langle X, \leq\rangle$ is a directed complete partial order *(DCPO) iff, for every directed set D, $\sup D$ exists; furthermore, $\langle X, \leq\rangle$ is called* pointed *if there exists an element $\bot \in X$ such that $\bot \leq x$ for all $x \in X$.*

▶ **Definition 3.5** (Scott Continuity)**.** *Given two DCPOs $\langle X, \leq_X\rangle$ and $\langle Y, \leq_Y\rangle$, a function $f\colon X \to Y$ is* Scott Continuous *if it is monotone: $f(x) \leq_Y f(x')$ for all $x \leq_X x'$; and preserves suprema of directed sets: $\sup_{x\in D} f(x) = f(\sup D)$ for all $D \subseteq X$ directed.*

▶ **Lemma 3.6.** *For any directed set $D \subseteq \mathit{lpof}(L)$, $\sup D = \langle N, <, \lambda, \varphi\rangle$, where:*

$$N \triangleq \bigcup_{\beta\in D} N_\beta \qquad < \triangleq \bigcup_{\beta\in D} <_\beta \qquad \lambda(x) \triangleq \sup_{\beta\in D\,:\,x\in N_\beta} \lambda_\beta(x) \qquad \varphi(x) \triangleq \psi_x$$

and $\psi_x = \varphi_\beta(x)$, for all $\beta \in D$ such that $x \in N_\beta$.

In Lemma 3.6, φ is well defined since, for every $x \in \mathit{nodes}$ and $\beta, \beta' \in D$ that contain x, it must be that $\varphi_\beta(x) = \varphi_{\beta'}(x)$. Indeed, since D is directed, there must exist a $\gamma \in D$ such that $\beta, \beta' \sqsubseteq_{\mathit{lpof}} \gamma$ and, by Definition 3.2(3b), we have that $\varphi_\beta(x) = \varphi_\gamma(x) = \varphi_{\beta'}(x)$.

Notice that $\mathit{lpof}(L)$ is *not* pointed, since the node at the root can vary; hence, there is no single LPOF that is smaller (w.r.t. $\sqsubseteq_{\mathit{lpof}}$) than all other LPOFs. This situation will change when moving to pomsets [67, Lemma D.6], which abstract away from the specific nodes.

3.2 Pomsets with Formulae

Although LPOFs express causality and branching behavior, they are not ideal semantic structures because information about node identifiers is extraneous. We instead work with *partially ordered multisets* (pomsets), which abstract away from the specific names used.

▶ **Definition 3.7** (LPOF Isomorphism)**.** *Let α and β be LPOFs and $f\colon N_\alpha \to N_\beta$ be a bijection between their nodes. Define $f(\alpha) = \langle N, <, \lambda, \varphi\rangle$ as*

$$N \triangleq \{f(x) \mid x \in N_\alpha\} \quad < \triangleq \{(f(x), f(y)) \mid x <_\alpha y\} \quad \lambda \triangleq \lambda_\alpha \circ f^{-1} \quad \varphi \triangleq f \circ \varphi \circ f^{-1}$$

where $f(\psi)$ syntactically renames the variables of ψ in the obvious way; α and β are isomorphic, written $\alpha \equiv \beta$, iff there is a bijection $f\colon N_\alpha \to N_\beta$ such that $f(\alpha) = \beta$.

▶ **Definition 3.8** (Pomsets with Formulae). *Let* $[\alpha] \triangleq \{\beta \in \mathit{lpof}(L) \mid \alpha \equiv \beta\}$ *be the isomorphism class of* α*. A pomset with formulae (or, simply, pomset)* $\boldsymbol{\alpha} \in \mathit{pom}(L)$ *is an isomorphism class of* LPOFs*:* $\mathit{pom}(L) \triangleq \{[\alpha] \mid \alpha \in \mathit{lpof}(L)\}$*. Let* $\mathit{pom}_{\mathsf{fin}}(L)$ *be the set of pomsets with finitely many nodes. Finally, let* $\boldsymbol{\alpha} \sqsubseteq_{\mathit{pom}} \boldsymbol{\beta}$ *iff* $\forall \alpha \in \boldsymbol{\alpha}.\, \exists \beta \in \boldsymbol{\beta}.\, \alpha \sqsubseteq_{\mathit{lpof}} \beta$*.*

The semantics of programs depends on a variety of pomset composition operations (sequential, guarded, and parallel composition). We will also later *linearize* pomsets into state transformer functions. Some of these operations cannot be defined inductively on infinite structures, so we need ways to extend functions on finite structures to infinite ones. We first show that infinite pomsets correspond to the suprema of their finite approximations.

▶ **Lemma 3.9** (Finite Approximations). *For any* $\boldsymbol{\alpha} \in \mathit{pom}(L)$*,* $\boldsymbol{\alpha} = \sup \lfloor \boldsymbol{\alpha} \rfloor_{\mathsf{fin}}$*, where* $\lfloor \boldsymbol{\alpha} \rfloor_{\mathsf{fin}} \triangleq \{\boldsymbol{\beta} \in \mathit{pom}_{\mathsf{fin}}(L) \mid \boldsymbol{\beta} \sqsubseteq_{\mathit{pom}} \boldsymbol{\alpha}\}$ *is the set of finite approximations of* $\boldsymbol{\alpha}$*.*

Finite approximations coincide with the approximation order $\ll$ of [1] instantiated to pomsets (see [67, Appendix E] for more details); thus, from now on, we let $\boldsymbol{\alpha} \ll \boldsymbol{\beta}$ denote $\boldsymbol{\alpha} \in \lfloor \boldsymbol{\beta} \rfloor_{\mathsf{fin}}$. We are now ready to show that every operation f on finite pomsets can be extended to infinite ones by defining the operation on $\boldsymbol{\alpha}$ as the sup of the images through f of all the approximations of $\boldsymbol{\alpha}$. This will be useful in the remainder of the paper.

▶ **Lemma 3.10** (Extension). *Let* $f \colon \mathit{pom}_{\mathsf{fin}}(L)^n \to T$ *be a monotone function on the* DCPO *$\langle T, \leq \rangle$. Then* $f^* \colon \mathit{pom}(L)^n \to T$*, shown below, is well-defined and Scott continuous:*

$$f^*(\boldsymbol{\alpha}_1, \ldots, \boldsymbol{\alpha}_n) \triangleq \sup_{\boldsymbol{\alpha}'_1 \ll \boldsymbol{\alpha}_1} \cdots \sup_{\boldsymbol{\alpha}'_n \ll \boldsymbol{\alpha}_n} f(\boldsymbol{\alpha}'_1, \ldots, \boldsymbol{\alpha}'_n)$$

4 Pomset Operations

In this section, we define pomset operations that mirror the program syntax $C \in \mathit{cmd}$ from (1). We first define the singleton LPOF $\langle \ell \rangle_x$ on node x with label $\ell \in L$; this generalizes to $\langle \ell \rangle$ on pomsets and will be used to give the semantics to **skip** and atomic actions:

$$\langle \ell \rangle_x \triangleq \langle \{x\}, \emptyset, [x \mapsto \ell], [x \mapsto \mathsf{true}] \rangle \in \mathit{lpof}(L) \qquad \langle \ell \rangle \triangleq \{\langle \ell \rangle_x \mid x \in \mathit{nodes}\} \in \mathit{pom}(L)$$

4.1 Guarded Choice

We now define guard, recording the causality between a test and the two branches of computation defined on it. Assuming that $N_\alpha \cap N_\beta = \emptyset$ and $x \notin N_\alpha \cup N_\beta$, we let $\mathsf{guard}(x, \ell, \alpha, \beta) = \langle N, <, \lambda, \varphi \rangle$ where we overload $\wedge$ s.t. $\psi \wedge \mathsf{true} = \mathsf{true} \wedge \psi = \psi$, and:

$$N \triangleq \{x\} \cup N_\alpha \cup N_\beta \qquad < \triangleq <_\alpha \cup <_\beta \cup (\{x\} \times (N_\alpha \cup N_\beta))$$

$$\lambda(y) \triangleq \begin{cases} \ell & \text{if } y = x \\ \lambda_\alpha(y) & \text{if } y \in N_\alpha \\ \lambda_\beta(y) & \text{if } y \in N_\beta \end{cases} \qquad \varphi(y) \triangleq \begin{cases} \mathsf{true} & \text{if } y = x \\ \varphi_\alpha(y) \wedge x & \text{if } y \in N_\alpha \\ \varphi_\beta(y) \wedge \neg x & \text{if } y \in N_\beta \end{cases}$$

So, $\mathsf{guard}(x, b, \alpha, \beta)$ joins α and β with a new root node x, whose label is b, and additionally, the formulae in α and β are updated to require that b passes or fails, respectively. As an example, $\mathsf{guard}(x_{b_1}, b_1, \langle a_1 \rangle_{x_{a_1}}, \mathsf{guard}(x_{b_2}, b_2, \langle a_2 \rangle_{x_{a_2}}, \langle a_3 \rangle_{x_{a_3}}))$ produces the LPOF depicted in (4). For finite pomsets, we define the guard operation as follows:

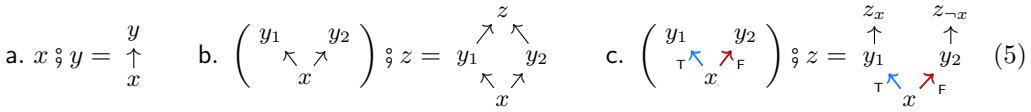
$$\mathsf{guard}(\ell, \boldsymbol{\alpha}, \boldsymbol{\beta}) \triangleq \{\mathsf{guard}(x, \ell, \alpha, \beta) \mid \alpha \in \boldsymbol{\alpha},\ \beta \in \boldsymbol{\beta}, N_\alpha \cap N_\beta = \emptyset, x \notin N_\alpha \cup N_\beta\}$$

Since guard is monotone [67, Lemma F.8], we can extend it to infinite pomsets:

$$\mathsf{guard}(\ell, \boldsymbol{\alpha}, \boldsymbol{\beta}) \triangleq \sup_{\boldsymbol{\alpha}' \ll \boldsymbol{\alpha}} \sup_{\boldsymbol{\beta}' \ll \boldsymbol{\beta}} \mathsf{guard}(\ell, \boldsymbol{\alpha}', \boldsymbol{\beta}')$$

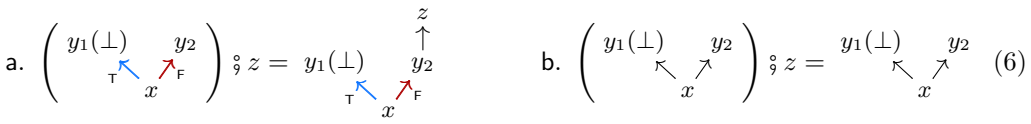
4.2 Sequential Composition

The sequential composition operation $\boldsymbol{\alpha} \fatsemi \boldsymbol{\beta}$ is meant to enforce that all actions in $\boldsymbol{\alpha}$ must occur before any of those in $\boldsymbol{\beta}$. This causality dependency interacts with guarded branching, as we will now see, which makes the formal definition of $\fatsemi$ more challenging. Consider the following LPOFs (where we denote with x the singleton $\langle \ell \rangle_x$ when the label is not relevant):



(5)

The sequential composition of two singletons (5a) simply creates a causality between them. When the program has forked into parallel (unguarded) branches (5b), $\fatsemi$ introduces a diamond structure, as is typical in pomset semantics. However, if the program has a *guarded* branch (5c), then the following actions must be copied after each branch. Copying is essential to correctly account for loops: in the program (**while** b **do** a) ; a', if we only created a single node labelled with a', then that node would not be finitely preceded, invalidating Lemma 3.9; see Figure 2 and [67, Appendix B.2] for a more detailed account. When a structure contains $\bot$ nodes, so that some paths are *stuck*, the behavior of $\fatsemi$ is more complicated. Consider the following LPOFs, where node y_1 is labelled with $\bot$ and all other nodes have non-$\bot$ labels:



(6)

When composing the singleton z after a guarded branch with one stuck path (6a), z is only added to the non-stuck branch. In the program execution, only one branch will be taken, so z can be safely executed as long at the test x is false. However, the same is not true for parallel branching; in (6b), both y_1 and y_2 will eventually be scheduled, so z must occur after y_1 – which is equivalent to not occurring at all – thus it does not appear in the final structure. This behavior ensures monotonicity of $\fatsemi$ [67, Lemma F.5].

We first define $\fatsemi$ for finite LPOFs, and then extend our definition to infinite ones using Lemma 3.10. In the rest of this section, we assume that $\alpha, \beta \in \mathit{lpof}_{\mathsf{fin}}(L)$. We start by defining the notion of stuck computations, extensible nodes, and branches.

$$\mathsf{stuck}_\alpha \triangleq \bigvee_{x \in \mathsf{Bot}_\alpha} \varphi_\alpha(x) \qquad \mathsf{ext}_\alpha \triangleq \{x \in N_\alpha \mid \varphi_\alpha(x) \not\Rightarrow \mathsf{stuck}_\alpha\}$$

$$\mathsf{br}_\alpha \triangleq \left\{ \varphi_\alpha(S) \;\middle|\; \begin{array}{l} \emptyset \subset S \subseteq \mathsf{ext}_\alpha,\ \varphi_\alpha(S) \Rightarrow \neg\mathsf{stuck}_\alpha, \\ \forall T.\ S \subset T \subseteq \mathsf{ext}_\alpha \Rightarrow \neg\mathsf{sat}(\varphi_\alpha(T)) \end{array} \right\}$$

The formula stuck_α indicates which nodes are guaranteed to encounter a $\bot$ later in their execution. Extensible nodes $x \in \mathsf{ext}_\alpha$ are not stuck, so there is some computation path that they can take without encountering any $\bot$ node. In (5), all nodes are extensible, whereas in (6a) only x and y_2 are extensible and in (6b) none are extensible. The set of branches br_α contains all of the maximal formulae that can be obtained as the conjunction of formulae of extensible nodes: there, for a set $S \subseteq N_\alpha$, we define $\varphi_\alpha(S) \triangleq \bigwedge_{x \in S} \varphi_\alpha(x)$. For example, in (5a,b), $\mathsf{br} = \{\mathsf{true}\}$ since there are no tests; in (5c), $\mathsf{br} = \{x, \neg x\}$ since both outcomes of the test are not stuck; in (6a), $\mathsf{br} = \{\neg x\}$ since the program is stuck if x passes; and in (6b), $\mathsf{br} = \emptyset$, since all paths are stuck. The set of branches increases monotonically with $\sqsubseteq_{\mathit{lpof}}$. Refer to [67, Appendix B.1] for more examples of extensibility and branches. As we saw in (5c), we will need an isomorphic copy $\beta_\psi \equiv \beta$ for each branch $\psi \in \mathsf{br}_\alpha$. These copies are

generated by a function $f\colon \mathsf{br}_\alpha \to [\beta]$ such that the nodes of $f(\psi)$ are disjoint from those of α and of every $f(\psi')$ where $\psi \neq \psi'$. The function f is drawn from the following set:

$$\mathsf{copy}_{\alpha,\beta} \triangleq \left\{ f\colon \mathsf{br}_\alpha \to [\beta] \;\middle|\; \forall \psi \in \mathsf{br}_\alpha.\, (N_{f(\psi)} \cap N_\alpha = \emptyset \;\wedge\; \forall \psi' \neq \psi. N_{f(\psi)} \cap N_{f(\psi')} = \emptyset) \right\}$$

Given any $\alpha, \beta \in \mathit{lpof}_{\mathsf{fin}}(L)$ and $f \in \mathsf{copy}_{\alpha,\beta}$, we define sequential composition as $\alpha \fatsemi_f \beta = \langle N, <, \lambda, \varphi \rangle$ where $\beta_\psi \triangleq f(\psi)$ and the components are defined as follows:

$$N \triangleq N_\alpha \cup \bigcup_{\psi \in \mathsf{br}_\alpha} N_{\beta_\psi} \qquad < \triangleq <_\alpha \cup \bigcup_{\psi \in \mathsf{br}_\alpha} \left(<_{\beta_\psi} \cup \left(\{x \in N_\alpha \mid \psi \Rightarrow \varphi_\alpha(x)\} \times N_{\beta_\psi} \right) \right)$$

$$\lambda(x) \triangleq \begin{cases} \lambda_\alpha(x) & \text{if } x \in N_\alpha \\ \lambda_{\beta_\psi}(x) & \text{if } x \in N_{\beta_\psi} \end{cases} \qquad \varphi(x) \triangleq \begin{cases} \varphi_\alpha(x) & \text{if } x \in N_\alpha \\ \varphi_{\beta_\psi}(x) \wedge \psi & \text{if } x \in N_{\beta_\psi} \end{cases}$$

(where, again, $\psi \wedge \mathsf{true} = \mathsf{true} \wedge \psi = \psi$). So, the nodes of $\alpha \fatsemi_f \beta$ are the nodes of α, and all the nodes of the isomorphic copies β_ψ. The new order preserves the causalities in α and in each β_ψ, and additionally requires that β_ψ occurs after all the nodes in the branch ψ. All labels are preserved, and formulae in β_ψ are updated to also include ψ. Sequential composition for finite pomsets is defined below (left), and is defined for infinite pomsets by extension (right).

$$\boldsymbol{\alpha} \fatsemi \boldsymbol{\beta} \triangleq \{\alpha \fatsemi_f \beta \mid \alpha \in \boldsymbol{\alpha},\ \beta \in \boldsymbol{\beta},\ f \in \mathsf{copy}_{\alpha,\beta}\} \qquad \boldsymbol{\alpha} \fatsemi \boldsymbol{\beta} \triangleq \sup_{\boldsymbol{\alpha}' \ll \boldsymbol{\alpha}} \sup_{\boldsymbol{\beta}' \ll \boldsymbol{\beta}} \boldsymbol{\alpha}' \fatsemi \boldsymbol{\beta}'$$

▶ Remark 4.1. We can now explain Condition (2c) in Definition 3.1, requiring single-rootedness, which may seem strange in a framework with concurrency; indeed, the usual semantics of two commands put in parallel is obtained by taking the (disjoint) union of their pomsets (and this yields several possible minimum elements in the resulting pomset). However, having multi-rooted LPOFs would make sequential composition not monotone. For example, $y \sqsubseteq_{\mathit{lpof}} y\ z$ but $x \fatsemi y \triangleq \begin{smallmatrix} & y \\ & \uparrow \\ x & \end{smallmatrix} \not\sqsubseteq_{\mathit{lpof}} \begin{smallmatrix} y & & z \\ & \nwarrow\ \nearrow & \\ & x & \end{smallmatrix} \triangleq x \fatsemi (y\ z)$. This is undesirable and easily fixed with the mild requirement of single-rootedness in Definition 3.1.

4.3 Parallel Composition

We finally define parallel composition. We assume that L contains a special (non-$\bot$) label fork that denotes thread forking and let $\mathsf{nFork}_\alpha \triangleq N_\alpha \setminus \{x \in \min_\alpha \mid \lambda_\alpha(x) = \mathsf{fork}\}$ denote the set of nodes of α without its fork-minimal elements (if any). Then, for both finite and infinite LPOFs α and β, and for any $x \notin N_\alpha \cup N_\beta$, we let $\alpha \parallel_x \beta \triangleq \langle N, <, \lambda, \varphi \rangle$, where

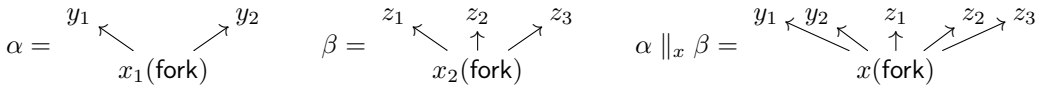
$$N \triangleq \{x\} \cup \mathsf{nFork}_\alpha \cup \mathsf{nFork}_\beta$$

$$< \triangleq (\{x\} \times (\mathsf{nFork}_\alpha \cup \mathsf{nFork}_\beta)) \cup (<_\alpha \cap (\mathsf{nFork}_\alpha \times \mathsf{nFork}_\alpha)) \cup (<_\beta \cap (\mathsf{nFork}_\beta \times \mathsf{nFork}_\beta))$$

$$\lambda(y) \triangleq \begin{cases} \mathsf{fork} & \text{if } y = x \\ \lambda_\alpha(y) & \text{if } y \in \mathsf{nFork}_\alpha \\ \lambda_\beta(y) & \text{if } y \in \mathsf{nFork}_\beta \end{cases} \qquad \varphi(y) \triangleq \begin{cases} \mathsf{true} & \text{if } y = x \\ \varphi_\alpha(y) & \text{if } y \in \mathsf{nFork}_\alpha \\ \varphi_\beta(y) & \text{if } y \in \mathsf{nFork}_\beta \end{cases}$$

Notice that the branching that arises from $\parallel$ is conceptually (and practically) different from the branching that arises from a test: the branching due to $\parallel$ does not exclude any branch, whereas the two branches due to a test are mutually exclusive. This is apparent by the different way in which we handle the formulae associated to the nodes after the branch: they remain the same under parallel composition, whereas they are extended with a new conjunct (specifying the value of the test at the branch) under a guard. The complication arising in our handling of parallel composition through fork is that we want one single node labelled with fork, even if we put in parallel many LPOFs.

For example, by putting in parallel the singleton LPOFs made up, respectively, by nodes y_1 and y_2 and by nodes z_1, z_2 and z_3 (and where fork is the label of the root nodes, say x_1 and x_2 respectively), we obtain the LPOFs α, β, and their parallel composition as follows:



where x_1 and x_2 disappear and the new root is x, labelled fork.

As a side note, observe that the way in which we defined parallel composition entails that $\alpha \sqsubseteq_{lpof} \alpha \parallel_x \beta$ if and only if $\alpha = \langle\bot\rangle_x$: the LPOF α has just a node x labelled with $\bot$; the second LPOF has node x labelled with fork that precedes a node labelled with $\bot$ and all the nodes of β. Also, a somewhat unusual notion of associativity for parallel of LPOFs holds, viz. $(\alpha \parallel_x \beta) \parallel_y \gamma = \alpha \parallel_y (\beta \parallel_z \gamma)$; the expectable notion of associativity is recovered for pomsets, where we abstract from the specific node set[3]. Like before, we define parallel composition for (both finite and infinite) pomsets on top of that for LPOFs:

$$\boldsymbol{\alpha} \parallel \boldsymbol{\beta} \triangleq \{\alpha \parallel_x \beta \mid \alpha \in \boldsymbol{\alpha},\ \beta \in \boldsymbol{\beta},\ x \notin N_\alpha \cup N_\beta\}$$

▶ Remark 4.2. It is impossible for $\parallel$ to be *both* associative *and* Scott continuous. If it were continuous, then $f(\boldsymbol{\alpha}) \triangleq \langle\ell\rangle \parallel \boldsymbol{\alpha}$ would have a least fixed point $\mathsf{lfp}(f) = \langle\ell\rangle \parallel (\langle\ell\rangle \parallel \cdots)$, and, due to associativity, this pomset is infinitely branching, which invalidates Lemma 3.9.

5 Denotational Semantics

We now define denotational semantics for commands $C \in \mathit{cmd}$ (see (1)) based on *pom*. Although actions at this stage are uninterpreted, *pom* allows for actions with a variety of different *computational effects*, which we will see in Section 5.2. Those actions $a \in \mathit{act}$ are drawn from a DCPO $\langle \mathit{act}, \leq_{\mathit{act}} \rangle$ and tests $b \in \mathit{test}$ are drawn from a set *test* (with a flat order), which is closed under conjunction, disjunction, and negation. We first define a compositional denotational model for this language in Section 5.1, and then in Section 5.2 we show how to interpret pomsets as state transformers, to learn their behavior on particular inputs.

5.1 Pomset Semantics

We will now discuss the semantics of programs $C \in \mathit{cmd}$ as pomsets, which record both the causality between atomic actions and the branching behavior of tests. The label set $\mathit{label} \triangleq \mathit{act} \cup \mathit{test} \cup \{\mathsf{fork}, \bot\}$ consists of actions, tests, fork, and bottom nodes, which forms a pointed DCPO, with $\bot$ as the bottom. Actions are ordered according to $\leq_{\mathit{act}}$; for any other label $\ell \in \mathit{test} \cup \{\mathsf{fork}\}$, we only have $\bot \leq \ell$ and $\ell \leq \ell$. We will also henceforth use $\mathit{pom} \triangleq \mathit{pom}(\mathit{label})$ to refer to pomsets over this label set.

The semantic function $[\![-]\!] : \mathit{cmd} \to \mathit{pom}$, mapping commands to pomsets, is shown on the left of Figure 2. Its definition is straightforward using the pomset operations; in particular, **skip** is interpreted as a singleton fork and while-loops are interpreted as the least fixed point of the characteristic function $\Phi_{\langle C,b\rangle} : \mathit{pom} \to \mathit{pom}$ defined as follows:

$$\Phi_{\langle C,b\rangle}(\boldsymbol{\alpha}) \triangleq \mathsf{guard}(b, [\![C]\!] \fatsemi \boldsymbol{\alpha}, [\![\mathbf{skip}]\!])$$

[3] We remark that $\fatsemi$ on LPOFs also satisfies a non-straightforward associativity, that however becomes the usual one when passing to pomsets.

$$
\begin{aligned}
[\![\textbf{skip}]\!] &\triangleq \langle \mathsf{fork} \rangle \\
[\![a]\!] &\triangleq \langle a \rangle \\
[\![C_1; C_2]\!] &\triangleq [\![C_1]\!] \fatsemi [\![C_2]\!] \\
[\![C_1 \mid C_2]\!] &\triangleq [\![C_1]\!] \parallel [\![C_2]\!] \\
[\![\textbf{if } b \textbf{ then } C_1 \textbf{ else } C_2]\!] &\triangleq \mathsf{guard}(b, [\![C_1]\!], [\![C_2]\!]) \\
[\![\textbf{while } b \textbf{ do } C]\!] &\triangleq \mathsf{lfp}\left(\Phi_{\langle C,b\rangle}\right)
\end{aligned}
$$

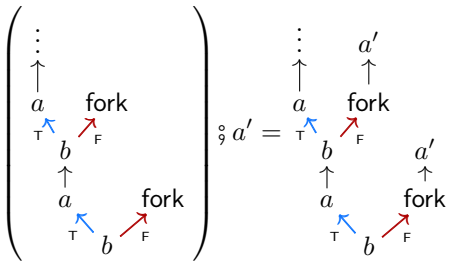
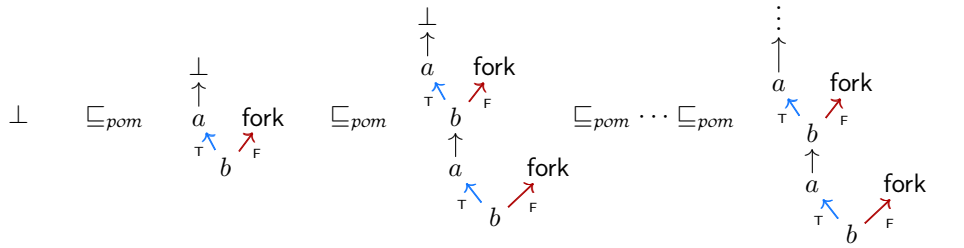


Figure 2 Left: pomset semantics of commands $[\![-]\!] : \mathit{cmd} \to \mathit{pom}$. Right: semantics of the program $(\textbf{while } b \textbf{ do } a)\ ;\ a'$.

Clearly $\Phi_{\langle C,b\rangle}$ is Scott continuous, as it is defined by Scott continuous operations guard and $\fatsemi$ [67, Corollaries F.6 and F.9]. So, by Kleene's fixed point theorem, $\Phi_{\langle C,b\rangle}$ has a least fixed point given by:

$$[\![\textbf{while } b \textbf{ do } C]\!] = \mathsf{lfp}\left(\Phi_{\langle C,b\rangle}\right) = \sup_{n\in\mathbb{N}} \Phi^n_{\langle C,b\rangle}\left(\bot_{\mathit{pom}}\right)$$

where $f^0 \triangleq \mathsf{id}$ and $f^{n+1} \triangleq f \circ f^n$. Considering a loop $[\![\textbf{while } b \textbf{ do } a]\!]$ with a single action a as its body, we can unroll the definition several times to get the following infinite chain:



$\Phi^0_{\langle a,b\rangle}(\bot_{\mathit{pom}})$ $\Phi^1_{\langle a,b\rangle}(\bot_{\mathit{pom}})$ $\Phi^2_{\langle a,b\rangle}(\bot_{\mathit{pom}})$ $\sup_{n\in\mathbb{N}} \Phi^n_{\langle a,b\rangle}(\bot_{\mathit{pom}})$

The supremum of this chain is clearly an infinite structure with a terminating branch for each $n \in \mathbb{N}$, and an infinitely ascending spine. The $\bot$ node is pushed to a progressively higher level after each unrolling, so in the supremum it does not appear at all.

On the right of Figure 2, we show $[\![(\textbf{while } b \textbf{ do } a)\ ;\ a']\!]$, containing a loop with a single action in the body followed by another action. As discussed in Section 4.2, sequentially composing another action a' after a loop produces a copy of a' for each branch, resulting in countably many copies of a'. This approach is necessary to make a' finitely preceded.

The semantics obeys a binary branching property, guaranteeing that test nodes branch into exactly two successors, one where the test passes and another where it fails. Further, variables can only be introduced into formulae after a test. From now on, we assume that all pomsets have this property, which is clearly preserved by the operations of Section 4.

▶ **Definition 5.1** (Binary Branching). *A pomset $\boldsymbol{\alpha} \in \mathit{pom}$ has the* binary branching property *if, for every $\alpha \in \boldsymbol{\alpha}$ and every $x \in N_\alpha$ such that $\lambda_\alpha(x) \in \mathit{test}$, $\mathsf{succ}_\alpha(x) = \{y_1, y_2\}$ such that:*

1. $\varphi_\alpha(y_1) \Leftrightarrow \varphi_\alpha(x) \wedge x$;
2. $\varphi_\alpha(y_2) \Leftrightarrow \varphi_\alpha(x) \wedge \neg x$; *and*
3. $\mathsf{pred}_\alpha(y_1) = \mathsf{pred}_\alpha(y_2) = \{x\}$.

Furthermore, if x is not the successor of a test, then $\varphi_\alpha(x) \Leftrightarrow \bigwedge_{y\in\mathsf{pred}_\alpha(x)} \varphi_\alpha(y)$.

$$\mathsf{next}(\alpha,\psi,S) \triangleq \{x \in N_\alpha \setminus S \mid x{\downarrow_\alpha} \subseteq S, \psi \Rightarrow \varphi_\alpha(x)\}$$

$$\mathcal{L}_{\mathit{lpof}}(\alpha,\psi,S)(s) \triangleq \eta(s) \quad \text{if } \mathsf{next}(\alpha,\psi,S) = \emptyset$$

$$\mathcal{L}_{\mathit{lpof}}(\alpha,\psi,S)(s) \triangleq \mathop{\&}_{x\in\mathsf{next}(\alpha,\psi,S)} \begin{cases} \mathcal{L}_{\mathit{lpof}}\,(\alpha,\psi,S\cup\{x\})^\dagger\,([\![a]\!]_{\mathit{act}}\,(s)) & \text{if } \lambda_\alpha(x) = a \in \mathit{act} \\ \mathcal{L}_{\mathit{lpof}}(\alpha,\psi\wedge(\!|x = [\![b]\!]_{\mathit{test}}(s)|\!),S\cup\{x\})(s) & \text{if } \lambda_\alpha(x) = b \in \mathit{test} \\ \bot_D & \text{if } \lambda_\alpha(x) = \bot \\ \mathcal{L}_{\mathit{lpof}}(\alpha,\psi,S\cup\{x\})(s) & \text{if } \lambda_\alpha(x) = \mathsf{fork} \end{cases}$$

Figure 3 Linearization for finite LPOFs $\mathcal{L}_{\mathit{lpof}} : \mathit{lpof}_{\mathsf{fin}}(\mathit{label}) \times \mathit{form} \times \mathcal{P}(\mathit{nodes}) \to \mathcal{S} \to D(\mathcal{S})$.

5.2 Linearization

We now discuss how to produce a state transformer from the pomset semantics of Section 5.1, which is useful for understanding the input-output behavior of programs resulting from scheduling concurrent threads. This state transformer is obtained via *linearization*, which consists of considering all interleavings of the threads in order to interpret the semantics of a *complete* program as a function from inputs to collections of outputs. Once the program is linearized, more threads cannot be composed in parallel, as the causality information is gone. Nevertheless, linearization is useful to understand the program's behavior, and relate the behavior to other semantic models, as shown in Sections 2 and 6.

When performing linearization, we will interpret programs in a computational domain D, which must support nondeterminism – in order to model the different ways that threads can be interleaved – but may support additional computational effects too, which we will model using monads. Hence, we assume some familiarity with basic notions of monads and Kleisli categories; for a detailed introduction, we refer to [3]. More precisely, we need the following:

1. A domain D and set of states $\mathcal{S}$, such that $\langle D(\mathcal{S}), \sqsubseteq\rangle$ is a pointed DCPO with bottom $\bot_D$.
2. An associative, commutative, and monotone nondeterminism operator $\&\colon D(\mathcal{S})^2 \to D(\mathcal{S})$.
3. An additive monad in the category **dcpo** $\langle D, \eta, (-)^\dagger\rangle$, meaning that:
   a. the operations $\eta\colon X \to D(X)$ and $(-)^\dagger\colon (X \to D(Y)) \to D(X) \to D(Y)$ obey the monad laws $f^\dagger \circ \eta = f$, $\eta^\dagger = \mathsf{id}$, and $(g^\dagger \circ f)^\dagger = g^\dagger \circ f^\dagger$.
   b. Kleisli extension is Scott Continuous (*i.e.,* $\sup_{f\in D}\sup_{d\in D'} f^\dagger(d) = (\sup D)^\dagger(\sup D')$), strict (*i.e.,* $f^\dagger(\bot_D) = \bot_D$), and additive (*i.e.,* $f^\dagger(d \,\&\, d') = f^\dagger(d) \,\&\, f^\dagger(d')$) [66,69].
4. An interpretation function for actions $[\![-]\!]_{\mathit{act}} : \mathit{act} \to \mathcal{S} \to D(\mathcal{S})$ that is monotone (*i.e.,* $[\![a]\!]_{\mathit{act}}(s) \sqsubseteq [\![a']\!]_{\mathit{act}}(s)$ if $a \leq_{\mathit{act}} a'$) and one for tests $[\![-]\!]_{\mathit{test}} : \mathit{test} \to \mathcal{S} \to \mathbb{B}$.

There are many domains that obey these properties, depending on which *computational effects* are present. These include: the Hoare, Smyth, and Plotkin powerdomains for nondeterministic computation [1, 46, 53] (where the latter two additionally deal with nontermination), the convex powerset [20, 42] and the powerdomain of indexed valuations [60, 61, 63] (for mixed probabilistic and nondeterministic computation), and other domains for nondeterminism with exceptions [66, 69]. In Theorems 5.3 and 6.1, we will show that standard semantics in two of those domains are recovered from our pomset semantics.

We will start by defining a linearization operation on finite LPOFs, which is shown in Figure 3. Linearization is defined recursively, until there are no more nodes to schedule. It must be defined on finite structures, otherwise the recursion is not well-founded, but we can extend linearization to infinite structures using the extension lemma. In $\mathcal{L}_{\mathit{lpof}}(\alpha,\psi,S)$, the set $S \subseteq N_\alpha$ contains all the nodes that have already been processed and ψ is a path condition, indicating the outcomes of the tests associated to the nodes in S.

The function (α, ψ, S) gives the nodes that are ready to be scheduled, which includes all nodes that obey the path condition ψ, and whose predecessors are in S. If (α, ψ, S) is empty, then $\mathcal{L}_{lpof}(\alpha, \psi, S)$ simply returns the current state s using the monad unit η. If not, then it nondeterministically selects a next node, where $\&_{i \in I}\, d_i \triangleq d_{i_1} \,\&\, \cdots \,\&\, d_{i_n}$, for any finite index set $I = \{i_1, \ldots, i_n\}$. If the next node is an action, then it interprets the action using $[\![-]\!]_{act}$, and then uses Kleisli composition to compose the result with the linearization of the remaining LPOF. If the next node is a test, then that test is evaluated and the result is added to the path condition, where $(\!|x = 1|\!) \triangleq x$ and $(\!|x = 0|\!) \triangleq \neg x$. If the next node is $\bot$, then the linearization is $\bot_D$, and fork is treated like a no-op.

We next use $\mathcal{L}_{lpof}$ to define linearization on finite pomsets $\mathcal{L}_{\mathsf{fin}} \colon \mathit{pom}_{\mathsf{fin}}(\mathit{label}) \to \mathcal{S} \to D(\mathcal{S})$ as $\mathcal{L}_{\mathsf{fin}}([\alpha]) \triangleq \mathcal{L}_{lpof}(\alpha, \mathsf{true}, \emptyset)$. Clearly $\mathcal{L}_{lpof}(\alpha, \mathsf{true}, \emptyset) = \mathcal{L}_{lpof}(\beta, \mathsf{true}, \emptyset)$ if $\alpha \equiv \beta$, so $\mathcal{L}_{\mathsf{fin}}(\boldsymbol{\alpha})$ can be defined for any arbitrary representative LPOF $\alpha \in \boldsymbol{\alpha}$. Finally, linearization is extended to infinite pomsets $\mathcal{L} \colon \mathit{pom} \to \mathcal{S} \to D(\mathcal{S})$ by taking the supremum over all of the finite approximations: $\mathcal{L}(\boldsymbol{\alpha}) \triangleq \mathcal{L}^*_{\mathsf{fin}}(\boldsymbol{\alpha})$. Linearization over finite pomsets is monotone [67, Lemma G.3], therefore the extension is well-defined and Scott continuous.

To ensure that linearization acts as desired, we provide a sanity check lemma to relate the linearized semantics to well-known counterparts. Linearizing **skip** gives the monad unit; linearizing a singleton action has the same behavior as interpreting the action; linearizing a sequential composition is equal to Kleisli composition of the individual linearizations; linearizing an if-statement is equal to linearizing one of the two branches, depending on the truth of the guard; and the linearization of a while-loop is equal to the least fixed point over a different characteristic function $\Psi_{\langle f,b\rangle}$. The function $\Psi_{\langle f,b\rangle}$ inherits Scott continuity from the Kleisli composition operator, therefore the fixed point exists.

▶ **Lemma 5.2** (Linearization). *The following properties hold:*

$$\mathcal{L}([\![\mathbf{skip}]\!]) = \eta \qquad \mathcal{L}([\![a]\!]) = [\![a]\!]_{act} \qquad \mathcal{L}([\![C_1; C_2]\!]) = \mathcal{L}([\![C_2]\!])^\dagger \circ \mathcal{L}([\![C_1]\!])$$

$$\mathcal{L}([\![\mathbf{if}\ b\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2]\!])(s) = \begin{cases} \mathcal{L}([\![C_1]\!])(s) & \text{if } [\![b]\!]_{test}(s) = 1 \\ \mathcal{L}([\![C_2]\!])(s) & \text{if } [\![b]\!]_{test}(s) = 0 \end{cases}$$

$$\mathcal{L}([\![\mathbf{while}\ b\ \mathbf{do}\ C]\!]) = \mathsf{lfp}\left(\Psi_{\langle \mathcal{L}([\![C]\!]), b\rangle}\right) \text{ where } \Psi_{\langle f,b\rangle}(g)(s) \triangleq \begin{cases} g^\dagger(f(s)) & \text{if } [\![b]\!]_{test}(s) = 1 \\ \eta(s) & \text{if } [\![b]\!]_{test}(s) = 0 \end{cases}$$

Lemma 5.2 is quite significant; it implies that $\mathcal{L}([\![C]\!])$ corresponds to standard monadic semantics in a variety of domains. In particular, it brings us to our first adequacy theorem, showing that our pomset model recovers standard probabilistic semantics $[\![C]\!]_{\mathcal{C}} : \mathcal{S} \to \mathcal{C}(\mathcal{S})$ (shown in Figure 4 of Section A.2) [20, 36, 68] based on the convex powerset for a parallel-free fragment of our programming language.

▶ **Theorem 5.3.** *For any program $C \in \mathit{cmd}$ without parallel composition: $\mathcal{L}([\![C]\!]) = [\![C]\!]_{\mathcal{C}}$.*

Proof. By induction on the structure of C. The cases follow immediately from Lemma 5.2. ◀

6 Pomset Languages and Purely Nondeterministic Concurrency

Powerdomains give semantics to programs that combine nondeterminism with looping or recursion [46, 53]. In this section, we create a powerdomain instantiation of our semantics from Section 5 and prove that the resulting model corresponds to well-known models of non-probabilistic concurrent programs [27, 34]. We work in the *Hoare* or *lower* powerdomain, named as such for its connection to partial correctness and Hoare Logic [16, 21]. Indeed, the

Hoare powerdomain identifies the terminating traces of a program, but not that the program *always* terminates. From a concurrency perspective, this corresponds to *safety* properties [33]. For simplicity, we presume that the order over actions $\leq_{act}$ in this section is flat.

Since typical pomsets do not contain formulae [17, 18, 48], they cannot encode control flow branching introduced by if-statements and while-loops. As we already saw in Section 2, concurrent semantics typically use *pomset languages* – sets of pomsets – where each individual pomset contains only the actions that occur in a single branch of computation [27, 34].

We emulate pomset language semantics using pomsets with formulae. First, we define a label set $\mathit{label}_{\mathsf{PL}} \triangleq \mathit{act}_{\mathsf{PL}} \cup \mathit{test}_{\mathsf{PL}} \cup \{\mathsf{fork}, \bot\}$, whose components are as before, but now $\mathit{test}_{\mathsf{PL}} \triangleq \emptyset$, and instead $\mathit{act}_{\mathsf{PL}} \triangleq \mathit{act} \cup \{\textbf{assume}\ b \mid b \in \mathit{test}\}$ includes an action **assume** b for every test $b \in \mathit{test}$, which guarantees that a certain condition holds in the current branch. Pomset languages $\mathit{pom\mathcal{L}ang} \triangleq \mathcal{P}(\mathit{pom}_{\mathsf{fin}}(\mathit{label}_{\mathsf{PL}}))$ are sets of finite pomsets over $\mathit{label}_{\mathsf{PL}}$.

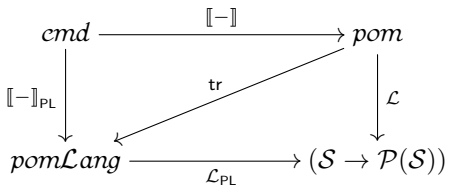
The pomset language semantics $[\![-]\!]_{\mathsf{PL}} : \mathit{cmd} \to \mathit{pom\mathcal{L}ang}$, which is standard [27, 34], is given in Figure 5 of Section A.3. In this model, each $\boldsymbol{\alpha} \in [\![C]\!]_{\mathsf{PL}}$ corresponds to a particular sequence of test resolutions. When an **assume** fails, the trace is eliminated. When a branch occurs, *i.e.,* in an if-statement or while-loop, the set of traces is duplicated to account for both the "true" and "false" branches, as we saw in Figure 1. Loops are interpreted as the (possibly infinite) union of all finite traces, so infinite pomsets are not needed.

The Hoare powerdomain consists of sets of states, ordered by subset inclusion $\langle \mathcal{P}(\mathcal{S}), \subseteq \rangle$, which is known to be a pointed DCPO with supremum given by union $\cup$ and $\emptyset$ as bottom. Let $\& \triangleq \cup$ and define the monad operations as $\eta(s) \triangleq \{s\}$ and $f^\dagger(S) \triangleq \bigcup_{s \in S} f(s)$, which are well known to be Scott continuous, strict, and additive [66]. Accordingly, action evaluation $[\![-]\!]_{\mathit{act}_{\mathsf{PL}}} : \mathit{act}_{\mathsf{PL}} \to \mathcal{S} \to \mathcal{P}(\mathcal{S})$ is given below, where **assume** actions result in $\{s\}$, if the test is true in state s, and $\emptyset$, otherwise. We also define a specialized linearization operation $\mathcal{L}_{\mathsf{PL}} : \mathit{pom\mathcal{L}ang} \to \mathcal{S} \to \mathcal{P}(\mathcal{S})$, which is simply a union over the linearization of all traces.

$$[\![a]\!]_{\mathit{act}_{\mathsf{PL}}}(s) \triangleq \begin{cases} [\![a]\!]_{\mathit{act}}(s) & \text{if } a \in \mathit{act} \\ \{s\} & \text{if } a = \textbf{assume}\ b,\ [\![b]\!]_{\mathit{test}}(s) = 1 \\ \emptyset & \text{if } a = \textbf{assume}\ b,\ [\![b]\!]_{\mathit{test}}(s) = 0 \end{cases} \qquad \mathcal{L}_{\mathsf{PL}}(S)(s) \triangleq \bigcup_{\boldsymbol{\alpha} \in S} \mathcal{L}_{\mathsf{fin}}(\boldsymbol{\alpha})(s)$$

Since all formulae are true, the next nodes to schedule are exactly those whose predecessors have all been processed, *i.e.,* $(\alpha, \mathsf{true}, S) = \{x \in N_\alpha \setminus S \mid x{\downarrow_\alpha} \subseteq S\}$. This corresponds to standard interleaving definitions of linearization [48]. We now prove that pomset language semantics $[\![-]\!]_{\mathsf{PL}}$ (Figure 5) corresponds exactly to the Hoare powerdomain instance of our semantic model from Section 5. The proof relies on a translation $\mathsf{tr} : \mathit{pom} \to \mathit{pom\mathcal{L}ang}$, which recovers a pomset language from a pomset with formulae, and is defined in [67, Appendix H].

▶ **Theorem 6.1** (Equivalence of Semantics). *The following diagram commutes:*



The upper commuting triangle does not depend on the Hoare powerdomain, so it is tempting to say that pomset languages give an adequate model in other domains too. However, as we showed in Section 2, just because a semantic structure exists does not mean that it conveys the desired meaning. Indeed, the fact that $\mathcal{L} = \mathcal{L}_{\mathsf{PL}} \circ \mathsf{tr}$ relies on two particular properties of the Hoare powerdomain. First, if $h(x) = f(x) \,\&\, g(x)$, then $h^\dagger(S) = f^\dagger(S) \,\&\, g^\dagger(S)$, which is

invalid in probabilistic domains. In fact, this corresponds to the problematic Equation (3) from Section 2. Second, $\mathcal{L}_{\mathsf{PL}}$ is computed as a union over a possibly infinite set (the union is infinite whenever the program contains a while-loop). Unbounded nondeterminism is known to cause problems in many domains – including the Smyth and Plotkin powerdomains [2,4,54] – thus no infinitary version of & exists.

So, although programs in any domain *can* be interpreted as pomset languages, those semantic objects may not give an adequate meaning to the program. Particularly in probabilistic domains – *i.e.,* the convex powerset – prior pomset language models [27,34] do not give us a way to find the relative probabilities of the different outcomes after actually running the program. Our new approach, using a single structure that records both causality and control flow, is more suitable for interpreting the semantics of concurrent programs in new domains, such as probabilistic computation. As we saw in Section 2 and Theorem 5.3, pomsets with formulae accurately capture the concurrent behavior of these programs, while also allowing us to recover the precise probabilities of each outcome.

7 Discussion and Related Work

We developed a new semantic structure – pomsets with formulae – and studied its domain-theoretic properties, which we used as basis to provide semantics for a probabilistic concurrent imperative language with unbounded loops. Our work builds on a rich history of using partial orders and pomsets in denotational models of concurrent programs [17,18,47], which have been used as a semantic basis for concurrent separation logic [8], concurrent Kleene algebras [22,26,27,34], and other applications [14,25,29].

While there are links between pomset semantics and operational models of concurrency [7], pomsets are often preferred for their finer notion of concurrency compared to the interleaving semantics offered by operational models. This is especially useful in the context of weak memory [25,29]. Probabilistic event structures model probabilistic process algebras [28,62,64,65], whereas we model a full imperative language with sequential composition, control flow, and unbounded loops, for which adding formulae was essential.

Concurrency has previously been combined with other computational effects in limited ways. For example, semantics for probabilistic concurrent programs have been defined in both operational [55,56] and denotational [70] styles, but these approaches are limited to programs with bounded looping constructs. The inclusion of unbounded looping adds significant complexity in the probabilistic case. Whereas finite-trace models are sufficient for many non-probabilistic scenarios, unbounded looping in probabilistic programs requires infinite traces, as the probability of termination may only become 1 in the limit. A proper theory of infinite traces requires pomsets to be enriched with a DCPO [37,38] or metric [9] structure; prior work in this area was very informative to our own development, although the inclusion of deterministic branching in our own pomset structure added significant complexity.

An operational model of probabilistic concurrency has been developed based on Markov Decision Processes [15]. This semantics is used to lower bound expected values, but does not straightforwardly extend to more complex domains such as the convex powerset, which give the full set of distributions over outcomes. In addition, a denotational model has been developed [44], where the underlying semantic structure is obtained as the solution to a domain equation. Our construction is more concrete, giving a full account of the causality in the program and thus capturing non-interleaving models of concurrency too.

Branching pomsets were recently introduced to model choices in choreographic programs [12–14]. While branching pomsets can, in theory, contain infinitely many nodes, the domain theoretic properties needed to approximate infinite structures have not been explored, which

was a significant focus of this paper. Indeed, our extension lemma (Lemma 3.10) was necessary for linearization, without which we would not have been able to relate our pomset model to the known convex powerset semantics [20, 36, 68].

Going forward, it would be interesting to explore the applicability of pomsets with formulae to convex powerdomain constructions other than Smyth [30, 57–59], or to other domains for mixing probabilities and nondeterminism including indexed valuations [60, 61, 63] and multisets of distributions [24, 32]. While our linearization procedure is based on an *omniscient* scheduler, we are also interested in exploring more restricted models including oblivious schedulers (which cannot see the outcomes of random sampling), and fair schedulers. This will be challenging, as those models are non-compositional and therefore linearization on infinite pomsets could not be defined via Lemma 3.10. Still, pomsets with formulae are a good basis for studying these questions, as the structure itself makes no assumptions about how scheduling of parallel branches occurs.

References

1 Samson Abramsky and Achim Jung. *Domain theory*, pages 1–168. Oxford University Press, Inc., USA, 1995.

2 Krzysztof Apt and Gordon Plotkin. Countable nondeterminism and random assignment. *J. ACM*, 33(4):724–767, August 1986. doi:10.1145/6490.6494.

3 Steve Awodey. *Category Theory*. Oxford Logic Guides. Ebsco Publishing, 2006.

4 Ralph-Johan Back. Semantics of unbounded nondeterminism. In Jaco de Bakker and Jan van Leeuwen, editors, *Automata, Languages and Programming*, pages 51–63, Berlin, Heidelberg, 1980. Springer Berlin Heidelberg. doi:10.1007/3-540-10003-2_59.

5 Gilles Barthe, Thomas Espitau, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. An Assertion-Based Program Logic for Probabilistic Programs. In *Programming Languages and Systems*, pages 117–144, Cham, 2018. Springer International Publishing. doi:10.1007/978-3-319-89884-1_5.

6 Filippo Bonchi, Ana Sokolova, and Valeria Vignudelli. Presenting Convex Sets of Probability Distributions by Convex Semilattices and Unique Bases. In Fabio Gadducci and Alexandra Silva, editors, *9th Conference on Algebra and Coalgebra in Computer Science (CALCO 2021)*, volume 211 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 11:1–11:18, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CALCO.2021.11.

7 Stephen Brookes. Traces, Pomsets, Fairness and Full Abstraction for Communicating Processes. In Luboš Brim, Mojmír Křetínský, Antonín Kučera, and Petr Jančar, editors, *CONCUR 2002 — Concurrency Theory*, pages 466–482, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. doi:10.1007/3-540-45694-5_31.

8 Stephen Brookes. A semantics for concurrent separation logic. In Philippa Gardner and Nobuko Yoshida, editors, *CONCUR 2004 - Concurrency Theory*, pages 16–34, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. doi:10.1007/978-3-540-28644-8_2.

9 J. W. de Bakker and J. H. A. Warmerdam. Metric pomset semantics for a concurrent language with recursion. In Irène Guessarian, editor, *Semantics of Systems of Concurrent Processes*, pages 21–49, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg. doi:10.1007/3-540-53479-2_2.

10 Jerry den Hartog. Verifying Probabilistic Programs Using a Hoare like Logic. In P. S. Thiagarajan and Roland Yap, editors, *Advances in Computing Science — ASIAN'99*, pages 113–125, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. doi:10.1007/3-540-46674-6_11.

11 Edsger W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Inf.*, 1(2):115–138, June 1971. doi:10.1007/BF00289519.

12 Luc Edixhoven and Sung-Shik Jongmans. Realisability of Branching Pomsets. In Silvia Lizeth Tapia Tarifa and José Proença, editors, *Formal Aspects of Component Software*, pages 185–204, Cham, 2022. Springer International Publishing. doi:10.1007/978-3-031-20872-0_11.

13 Luc Edixhoven, Sung-Shik Jongmans, José Proença, and Ilaria Castellani. Branching pomsets: Design, expressiveness and applications to choreographies. *Journal of Logical and Algebraic Methods in Programming*, 136:100919, 2024. doi:10.1016/j.jlamp.2023.100919.

14 Luc Edixhoven, Sung-Shik Jongmans, José Proença, and Guillermina Cledou. Branching Pomsets for Choreographies. *Electronic Proceedings in Theoretical Computer Science*, 365:37–52, August 2022. doi:10.4204/eptcs.365.3.

15 Ira Fesefeldt, Joost-Pieter Katoen, and Thomas Noll. Towards Concurrent Quantitative Separation Logic. In Bartek Klin, Sławomir Lasota, and Anca Muscholl, editors, *33rd International Conference on Concurrency Theory (CONCUR 2022)*, volume 243 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:24, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CONCUR.2022.25.

16 Robert W. Floyd. Assigning Meanings to Programs. In *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32, Providence, Rhode Island, 1967. American Mathematical Society. doi:10.1007/978-94-011-1793-7_4.

17 Jay L. Gischer. The equational theory of pomsets. *Theoretical Computer Science*, 61(2):199–224, 1988. doi:10.1016/0304-3975(88)90124-7.

18 Jan Grabowski. On Partial Languages. *Fundamenta Informaticae*, 4(2):427–498, 1981. doi:10.3233/fi-1981-4210.

19 Sergiu Hart, Micha Sharir, and Amir Pnueli. Termination of probabilistic concurrent program. *ACM Trans. Program. Lang. Syst.*, 5(3):356–380, July 1983. doi:10.1145/2166.357214.

20 Jifeng He, Karen Seidel, and Annabelle McIver. Probabilistic models for the guarded command language. *Science of Computer Programming*, 28(2):171–192, 1997. Formal Specifications: Foundations, Methods, Tools and Applications. doi:10.1016/S0167-6423(96)00019-6.

21 Charles Antony Richard Hoare. An Axiomatic Basis for Computer Programming. *Commun. ACM*, 12(10):576–580, October 1969. doi:10.1145/363235.363259.

22 Tony Hoare, Bernhard Möller, Georg Struth, and Ian Wehrman. Concurrent Kleene Algebra and its Foundations. *J. Log. Algebraic Methods Program.*, 80(6):266–296, 2011. doi:10.1016/J.JLAP.2011.04.005.

23 Bart Jacobs. Coalgebraic trace semantics for combined possibilitistic and probabilistic systems. *Electronic Notes in Theoretical Computer Science*, 203(5):131–152, 2008. Proceedings of the Ninth Workshop on Coalgebraic Methods in Computer Science (CMCS 2008). doi:10.1016/j.entcs.2008.05.023.

24 Bart Jacobs. From Multisets over Distributions to Distributions over Multisets. In *Proceedings of the 36th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '21, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1109/LICS52264.2021.9470678.

25 Radha Jagadeesan, Alan Jeffrey, and James Riely. Pomsets with preconditions: a simple model of relaxed memory. *Proc. ACM Program. Lang.*, 4(Oopsla), November 2020. doi:10.1145/3428262.

26 Peter Jipsen and M. Andrew Moshier. Concurrent Kleene algebra with tests and branching automata. *Journal of Logical and Algebraic Methods in Programming*, 85(4):637–652, 2016. Relational and algebraic methods in computer science. doi:10.1016/j.jlamp.2015.12.005.

27 Tobias Kappé, Paul Brunet, Alexandra Silva, Jana Wagemaker, and Fabio Zanasi. *Concurrent Kleene Algebra with Observations: From Hypotheses to Completeness*, pages 381–400. Springer International Publishing, 2020. doi:10.1007/978-3-030-45231-5_20.

28 Joost-Pieter Katoen. *Quantitative and Qualitative Extensions of Event Structures*. Phd thesis - research ut, graduation ut, University of Twente, Netherlands, 1996. doi:10.3990/1.9789036507998.

29 Ryan Kavanagh and Stephen Brookes. A Denotational Semantics for SPARC TSO. *Electronic Notes in Theoretical Computer Science*, 336:223–239, 2018. The Thirty-third Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXIII). doi:10.1016/j.entcs.2018.03.025.

30 Klaus Keimel and Gordon Plotkin. Mixed powerdomains for probability and nondeterminism. *Logical Methods in Computer Science*, Volume 13, Issue 1, January 2017. doi:10.23638/LMCS-13(1:2)2017.

31 Dexter Kozen. Semantics of probabilistic programs. In *20th Annual Symposium on Foundations of Computer Science (SFCS '79)*, pages 101–114, 1979. doi:10.1109/SFCS.1979.38.

32 Dexter Kozen and Alexandra Silva. Multisets and distributions. In Venanzio Capretta, Robbert Krebbers, and Freek Wiedijk, editors, *Logics and Type Systems in Theory and Practice: Essays Dedicated to Herman Geuvers on The Occasion of His 60th Birthday*, pages 168–187, Cham, 2024. Springer Nature Switzerland. doi:10.1007/978-3-031-61716-4_11.

33 L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, 1977. doi:10.1109/TSE.1977.229904.

34 Michael R. Laurence and Georg Struth. Completeness Theorems for Bi-Kleene Algebras and Series-Parallel Rational Pomset Languages. In Peter Höfner, Peter Jipsen, Wolfram Kahl, and Martin Eric Müller, editors, *Relational and Algebraic Methods in Computer Science*, pages 65–82, Cham, 2014. Springer International Publishing. doi:10.1007/978-3-319-06251-8_5.

35 Daniel Lehmann and Michael O. Rabin. On the advantages of free choice: a symmetric and fully distributed solution to the dining philosophers problem. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '81, pages 133–138, New York, NY, USA, 1981. Association for Computing Machinery. doi:10.1145/567532.567547.

36 Annabelle McIver and Carroll Morgan. *Abstraction, Refinement and Proof for Probabilistic Systems*. Monographs in Computer Science. Springer, 2005. doi:10.1007/b138392.

37 J. J. Ch. Meyer and E. P. de Vink. Pomset semantics for true concurrency with synchronization and recursion. In Antoni Kreczmar and Grazyna Mirkowska, editors, *Mathematical Foundations of Computer Science 1989*, pages 360–369, Berlin, Heidelberg, 1989. Springer Berlin Heidelberg. doi:10.1007/3-540-51486-4_83.

38 J. J. Ch. Meyer and E.P. de Vink. Applications of compactness in the smyth powerdomain of streams. *Theoretical Computer Science*, 57(2):251–282, 1988. doi:10.1016/0304-3975(88)90042-4.

39 Michael Mislove. Nondeterminism and probabilistic choice: Obeying the laws. In Catuscia Palamidessi, editor, *CONCUR 2000 — Concurrency Theory*, pages 350–365, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg. doi:10.1007/3-540-44618-4_26.

40 Michael Mislove, Joël Ouaknine, and James Worrell. Axioms for probability and nondeterminism. *Electronic Notes in Theoretical Computer Science*, 96:7–28, 2004. Proceedings of the 10th International Workshop on Expressiveness in Concurrency. doi:10.1016/j.entcs.2004.04.019.

41 Carroll Morgan, Annabelle McIver, and Karen Seidel. Probabilistic Predicate Transformers. *ACM Trans. Program. Lang. Syst.*, 18(3):325–353, May 1996. doi:10.1145/229542.229547.

42 Carroll Morgan, Annabelle McIver, Karen Seidel, and J. W. Sanders. Refinement-oriented probability for CSP. *Form. Asp. Comput.*, 8(6):617–647, November 1996. doi:10.1007/bf01213492.

43 Robert Morris. Counting large numbers of events in small registers. *Commun. ACM*, 21(10):840–842, October 1978. doi:10.1145/359619.359627.

44 Renato Neves. An adequacy theorem between mixed powerdomains and probabilistic concurrency, 2024. doi:10.48550/arXiv.2409.15920.

45 Louis Parlant. *Monad Composition via Preservation of Algebras*. PhD thesis, University College London, 2020. URL: https://discovery.ucl.ac.uk/id/eprint/10112228/.

46 Gordon Plotkin. A powerdomain construction. *SIAM Journal on Computing*, 5(3):452–487, 1976. doi:10.1137/0205035.

47 Vaughan R. Pratt. Semantical Considerations on Floyd-Hoare Logic. In *17th Annual Symposium on Foundations of Computer Science (sfcs 1976)*, pages 109–121, 1976. doi:10.1109/SFCS.1976.27.

48 Vaughan R. Pratt. Modeling concurrency with partial orders. *Int. J. Parallel Program.*, 15(1):33–71, 1986. doi:10.1007/BF01379149.

49 Michael O. Rabin. N-process synchronization by 4.log2n-valued shared variable. In *21st Annual Symposium on Foundations of Computer Science (sfcs 1980)*, pages 407–410, 1980. doi:10.1109/SFCS.1980.26.

50 Michael O. Rabin. The choice coordination problem. *Acta Inf.*, 17(2):121–134, June 1982. doi:10.1007/BF00288965.

51 Dana Scott. Outline of a Mathematical Theory of Computation. Technical Report PRG02, OUCL, November 1970.

52 Dana Scott and Christopher Strachey. Toward A Mathematical Semantics For Computer Languages. Technical Report Prg06, OUCL, August 1971.

53 Michael Smyth. Power domains. *Journal of Computer and System Sciences*, 16(1):23–36, 1978. doi:10.1016/0022-0000(78)90048-X.

54 Harald Søndergaard and Peter Sestoft. Non-determinism in Functional Languages. *The Computer Journal*, 35(5):514–523, October 1992. doi:10.1093/comjnl/35.5.514.

55 Joseph Tassarotti. *Verifying Concurrent Randomized Algorithms*. PhD thesis, Carnegie Mellon University, December 2018. URL: https://csd.cmu.edu/academics/doctoral/degrees-conferred/joseph-tassarotti.

56 Joseph Tassarotti and Robert Harper. A Separation Logic for Concurrent Randomized Programs. *Proc. ACM Program. Lang.*, 3(POPL), January 2019. doi:10.1145/3290377.

57 Regina Tix. *Continuous D-cones: convexity and powerdomain constructions*. PhD thesis, Darmstadt University of Technology, Germany, 1999. URL: https://d-nb.info/957239157.

58 Regina Tix. Convex power constructions for continuous d-cones. *Electronic Notes in Theoretical Computer Science*, 35:206–229, 2000. Workshop on Domains IV. doi:10.1016/S1571-0661(05)80746-7.

59 Regina Tix, Klaus Keimel, and Gordon Plotkin. Semantic domains for combining probability and non-determinism. *Electronic Notes in Theoretical Computer Science*, 222:3–99, 2009. doi:10.1016/j.entcs.2009.01.002.

60 Daniele Varacca. The powerdomain of indexed valuations. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 299–308, 2002. doi:10.1109/LICS.2002.1029838.

61 Daniele Varacca. *Probability, Nondeterminism and Concurrency: Two Denotational Models for Probabilistic Computation*. PhD thesis, University of Aarhus, November 2003. URL: https://www.brics.dk/DS/03/14/.

62 Daniele Varacca, Hagen Völzer, and Glynn Winskel. Probabilistic event structures and domains. *Theoretical Computer Science*, 358(2):173–199, 2006. Concurrency Theory (CONCUR 2004). doi:10.1016/j.tcs.2006.01.015.

63 Daniele Varacca and Glynn Winskel. Distributing probability over non-determinism. *Mathematical Structures in Computer Science*, 16(1):87–113, 2006. doi:10.1017/S0960129505005074.

64 Daniele Varacca and Nobuko Yoshida. Probabilistic π-Calculus and Event Structures. *Electronic Notes in Theoretical Computer Science*, 190(3):147–166, 2007. Proceedings of the Fifth Workshop on Quantitative Aspects of Programming Languages (QAPL 2007). doi:10.1016/j.entcs.2007.07.009.

65 Glynn Winskel. Probabilistic and Quantum Event Structures. In Franck van Breugel, Elham Kashefi, Catuscia Palamidessi, and Jan Rutten, editors, *Horizons of the Mind. A Tribute to Prakash Panangaden: Essays Dedicated to Prakash Panangaden on the Occasion*

of His 60th Birthday, pages 476–497, Cham, 2014. Springer International Publishing. doi:10.1007/978-3-319-06880-0_25.

66 Noam Zilberstein, Derek Dreyer, and Alexandra Silva. Outcome Logic: A Unifying Foundation for Correctness and Incorrectness Reasoning. *Proc. ACM Program. Lang.*, 7(OOPSLA1), April 2023. doi:10.1145/3586045.

67 Noam Zilberstein, Daniele Gorla, and Alexandra Silva. Denotational Semantics for Probabilistic and Concurrent Programs (Full Version), 2025. arXiv:2503.02768.

68 Noam Zilberstein, Dexter Kozen, Alexandra Silva, and Joseph Tassarotti. A demonic outcome logic for randomized nondeterminism. *Proc. ACM Program. Lang.*, 9(POPL), January 2025. doi:10.1145/3704855.

69 Noam Zilberstein, Angelina Saliling, and Alexandra Silva. Outcome separation logic: Local reasoning for correctness and incorrectness with computational effects. *Proc. ACM Program. Lang.*, 8(OOPSLA1), April 2024. doi:10.1145/3649821.

70 Noam Zilberstein, Alexandra Silva, and Joseph Tassarotti. Probabilistic Concurrent Reasoning in Outcome Logic: Independence, Conditioning, and Invariants, 2024. doi:10.48550/arXiv.2411.11662.

71 Maaike Zwart. *On the Non-Compositionality of Monads via Distributive Laws*. PhD thesis, University of Oxford, 2020. URL: https://ora.ox.ac.uk/objects/uuid:b2222b14-3895-4c87-91f4-13a8d046febb.

72 Maaike Zwart and Dan Marsden. No-Go Theorems for Distributive Laws. In *2019 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–13, 2019. doi:10.1109/lics.2019.8785707.

A Preliminaries and Omitted Definitions

A.1 Order Theory and Domain Theory

▶ **Definition A.1** (Poset). *A partially ordered set, or* poset*, is a set equipped with a partial order: that is, a pair $\langle X, \leq \rangle$ consisting of a set X and an order relation $\leq \subseteq X \times X$ which is reflexive, transitive and antisymmetric. When the order is instead irreflexive (*i.e., *$x \not< x$ for all $x \in X$*), the poset is called* strict *and the order relation is denoted by $<$.*

Given a strict poset $\langle X, < \rangle$ and $x \in X$, we define the upward and downward closures:

$$x{\uparrow} \triangleq \{y \in X \mid x < y\} \qquad\qquad x{\downarrow} \triangleq \{y \in X \mid y < x\};$$

and its set of immediate successors and predecessors as

$$\mathsf{succ}(x) \triangleq \{y \in x{\uparrow} \mid \nexists z.(x < z < y)\} \qquad \mathsf{pred}(x) \triangleq \{y \in x{\downarrow} \mid \nexists z.(y < z < x)\}$$

All the above operations can easily be extended to sets of elements, *e.g.,* $X{\uparrow} \triangleq \bigcup_{x \in X} x{\uparrow}$, for any set X. The set of minimal and maximal nodes of a poset is given by:

$$\max \triangleq \{x \in X \mid \mathsf{succ}(x) = \emptyset\} \qquad \min \triangleq \{x \in X \mid \mathsf{pred}(x) = \emptyset\}$$

To define our new structures we need a few more definitions on posets that will play a role in proving the existence of fixed points, essential for giving a semantics to unbounded loops. First, we recall the notion of downward closed sets.

▶ **Definition A.2** (Downward Closure). *Given a poset $\langle X, < \rangle$, a set $Y \subseteq X$ is downward closed, written $Y \subseteq_{\downarrow} X$, iff for all $y \in Y$ it holds that $y{\downarrow} \subseteq Y$.*

Second, we need a notion of a poset being finitely preceded.

▶ **Definition A.3** (Finitely Preceded). *A poset $\langle X, < \rangle$ is* finitely preceded *if there are finitely many elements smaller than each element; that is: $|x{\downarrow}| < \infty$, for all $x \in X$.*

Third, we define the level of elements in the poset.

▶ **Definition A.4** (Level [37]). *Given a finitely preceded poset $\langle X, <\rangle$, define* $\mathsf{lev}\colon X \to \mathbb{N}$*:*

$$\mathsf{lev}(x) \triangleq \sup\left\{n \;\middle|\; \exists x_0, \ldots, x_n \in X.\ x_0 \in \min\ \wedge\ x_n = x\ \wedge\ \forall i < n.\ x_{i+1} \in \mathsf{succ}(x_i)\right\}$$

We also define its inverse $\mathsf{lev}^{-1}\colon \mathbb{N} \to \mathcal{P}(X)$*:*

$$\mathsf{lev}^{-1}(n) \triangleq \{x \in X \mid \mathsf{lev}(x) = n\}$$

▶ **Definition A.5.** *The satisfaction relation $\vDash \subseteq (\mathit{nodes} \to \mathbb{B}) \times \mathit{form}$ for Boolean formulae is given below:*

$v \vDash \mathsf{true}$	*always*
$v \vDash \mathsf{false}$	*never*
$v \vDash \psi_1 \wedge \psi_2$	*iff* $v \vDash \psi_1$ *and* $v \vDash \psi_2$
$v \vDash \psi_1 \vee \psi_2$	*iff* $v \vDash \psi_1$ *or* $v \vDash \psi_2$
$v \vDash \neg\psi$	*iff* $v \nvDash \psi$
$v \vDash x$	*iff* $v(x) = 1$

A.2 The Convex Powerdomain

We begin by providing the omitted definitions from Section 2, regarding the DCPO structure of the convex powerset. We start by defining the order on probability distributions. For any set X, let $X_\perp = X \cup \{\perp\}$ and the order over distributions $\sqsubseteq_{\mathcal{D}} \subseteq \mathcal{D}(X_\perp) \times \mathcal{D}(X_\perp)$ be defined as follows:

$$\mu \sqsubseteq_{\mathcal{D}} \nu \quad \text{iff} \quad \forall x \in X.\ \mu(x) \leq \nu(x)$$

Note that this means that $\mu(\perp) \geq \nu(\perp)$ if $\mu \sqsubseteq_{\mathcal{D}} \nu$, and that the bottom of the order is $\perp_{\mathcal{D}} = [\perp \mapsto 1]$. A set of distributions $S \subseteq \mathcal{D}(X_\perp)$ is *upward closed* if, for all $\mu \in S$, it holds that $\mu \sqsubseteq_{\mathcal{D}} \nu$ implies $\nu \in S$. In addition, S is *Cauchy closed* if it is closed in the product of Euclidean topologies [36, Definition 5.4.3]. Now, the convex powerset is defined as follows:

$$\mathcal{C}(X) \triangleq \{S \subseteq \mathcal{D}(X_\perp) \mid S \text{ is nonempty, convex, upward closed, and Cauchy closed}\}$$

Convex powersets are ordered via the Smyth order [53] $\sqsubseteq_{\mathcal{C}} \subseteq \mathcal{C}(X) \times \mathcal{C}(X)$, defined below:

$$S \sqsubseteq_{\mathcal{C}} T \quad \text{iff} \quad \forall \nu \in T.\ \exists \mu \in S.\ \mu \sqsubseteq_{\mathcal{D}} \nu$$

Since the sets above are upward closed, the Smyth order collapses to reverse subset inclusion $S \sqsubseteq_{\mathcal{C}} T$ iff $S \supseteq T$. This makes the *convex powerdomain* $\langle \mathcal{C}(X), \sqsubseteq_{\mathcal{C}} \rangle$ a pointed DCPO with suprema given by set intersection and bottom $\perp_{\mathcal{C}} = \mathcal{D}(X_\perp)$ being the set of all distributions, for the full proof refer to [68].

Finally, we give the definitions for the monad operations [23]: unit $\eta\colon X \to \mathcal{C}(X)$ and Kleisli extension $(-)^\dagger\colon (X \to \mathcal{C}(Y)) \to \mathcal{C}(X) \to \mathcal{C}(Y)$.

$$\eta(x) \triangleq [x \mapsto 1] \qquad f^\dagger(S) \triangleq \left\{ \sum_{x \in \mathsf{supp}(\mu)} \mu(x) \cdot \nu_x \;\middle|\; \mu \in S, \forall x \in \mathsf{supp}(\mu).\ \nu_x \in f_\perp(x) \right\}$$

where $(-)_\perp\colon (X \to \mathcal{C}(Y)) \to X_\perp \to \mathcal{C}(Y)$ is defined as follows: $f_\perp(x) \triangleq f(x)$ if $x \in X$ and $f_\perp(\perp) \triangleq \perp_{\mathcal{C}}$ otherwise.

$$
\begin{aligned}
[\![\textbf{skip}]\!]_{\mathcal{C}}(s) &\triangleq \eta(s)\\
[\![C_1; C_2]\!]_{\mathcal{C}}(s) &\triangleq [\![C_2]\!]^{\dagger}_{\mathcal{C}}\left([\![C_1]\!]_{\mathcal{C}}(s)\right)\\
[\![\textbf{if } b \textbf{ then } C_1 \textbf{ else } C_2]\!]_{\mathcal{C}}(s) &\triangleq \begin{cases} [\![C_1]\!]_{\mathcal{C}}(s) & \text{if } [\![b]\!]_{\mathit{test}}(s) = 1\\ [\![C_2]\!]_{\mathcal{C}}(s) & \text{if } [\![b]\!]_{\mathit{test}}(s) = 0 \end{cases}\\
[\![\textbf{while } b \textbf{ do } C]\!]_{\mathcal{C}}(s) &\triangleq \mathsf{lfp}(\Psi_{\langle [\![C]\!]_{\mathcal{C}}, b\rangle})(s)\\
[\![a]\!]_{\mathcal{C}}(s) &\triangleq [\![a]\!]_{\mathit{act}}(s)
\end{aligned}
$$

Figure 4 Convex powerset semantics $[\![-]\!]_{\mathcal{C}} : \mathit{cmd} \to \mathcal{S} \to \mathcal{C}(\mathcal{S})$ due to [20, 36, 68], where Ψ is defined in Lemma 5.2.

$$
\begin{aligned}
[\![\textbf{skip}]\!]_{\mathsf{PL}} &\triangleq \{\langle \mathsf{fork}\rangle\}\\
[\![a]\!]_{\mathsf{PL}} &\triangleq \{\langle a\rangle\}\\
[\![C_1; C_2]\!]_{\mathsf{PL}} &\triangleq [\![C_1]\!]_{\mathsf{PL}} \fatsemi [\![C_2]\!]_{\mathsf{PL}}\\
[\![C_1 \mid C_2]\!]_{\mathsf{PL}} &\triangleq [\![C_1]\!]_{\mathsf{PL}} \parallel [\![C_2]\!]_{\mathsf{PL}}\\
[\![\textbf{if } b \textbf{ then } C_1 \textbf{ else } C_2]\!]_{\mathsf{PL}} &\triangleq (\{\langle \textbf{assume } b\rangle\} \fatsemi [\![C_1]\!]_{\mathsf{PL}}) \cup (\{\langle \textbf{assume } \neg b\rangle\} \fatsemi [\![C_2]\!]_{\mathsf{PL}})\\
[\![\textbf{while } b \textbf{ do } C]\!]_{\mathsf{PL}} &\triangleq \mathsf{lfp}\left(\Xi_{\langle C, b\rangle}\right)
\end{aligned}
$$

$$
\begin{aligned}
\text{where}\quad \Xi_{\langle C,b\rangle}(S) &\triangleq (\{\langle \textbf{assume } b\rangle\} \fatsemi [\![C]\!]_{\mathsf{PL}} \fatsemi S) \cup (\{\langle \textbf{assume } \neg b\rangle\} \fatsemi [\![\textbf{skip}]\!]_{\mathsf{PL}})\\
S \fatsemi T &\triangleq \{\boldsymbol{\alpha} \fatsemi \boldsymbol{\beta} \mid \boldsymbol{\alpha} \in S, \boldsymbol{\beta} \in T\}\\
S \parallel T &\triangleq \{\boldsymbol{\alpha} \parallel \boldsymbol{\beta} \mid \boldsymbol{\alpha} \in S, \boldsymbol{\beta} \in T\}
\end{aligned}
$$

Figure 5 Standard pomset language semantics $[\![-]\!]_{\mathsf{PL}} : \mathit{cmd} \to \mathit{pom}\mathcal{L}\mathit{ang}$.

As is required in Section 5.2, it is proven in [68] that the Kleisli extension for $\mathcal{C}$ defined above is Scott continuous and additive:

$$
\sup_{f \in D, x \in D'} f^{\dagger}(S) = (\sup D)^{\dagger}(\sup D') \qquad\qquad f^{\dagger}(S \,\&\, T) = f^{\dagger}(S) \,\&\, f^{\dagger}(T)
$$

Based on these operations, Figure 4 shows the convex powerset semantics for the parallel-free fragment of the language in (1), $[\![-]\!]_{\mathcal{C}} : \mathit{cmd} \to \mathcal{S} \to \mathcal{C}(\mathcal{S})$. This is precisely the same semantics used in prior work for reasoning about programs that are both nondeterministic and probabilistic [20, 36, 68]. This semantics does not capture parallel computation, which requires the more sophisticated semantic domains that we develop in this paper. The Scott continuity property above ensures that $\Psi_{\langle [\![C]\!]_{\mathcal{C}}, b\rangle} : (\mathcal{S} \to \mathcal{C}(\mathcal{S})) \to \mathcal{S} \to \mathcal{C}(\mathcal{S})$ is Scott continuous, and therefore the semantics of loops is well defined.

A.3 Pomset Languages

As we saw in Section 6, a pomset language is a set of finite pomsets without any tests or formulae. Control flow is determined by **assume** actions, which keep or eliminate traces depending on the whether the corresponding test passes or fails.

$$
\mathit{pom}\mathcal{L}\mathit{ang} \triangleq \mathcal{P}(\mathit{pom}_{\mathsf{fin}}(\mathit{label}_{\mathsf{PL}}))
$$

We give the pomset language semantics for a nondeterministic language in Figure 5.