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

- Luca Aceto (Reykjavik University)
- Susanne Albers (TU München)
- Chris Hankin (Imperial College London)
- Deepak Kapur (University of New Mexico)
- Michael Mitzenmacher (Harvard University)
- Madhavan Mukund (Chennai Mathematical Institute)
- Anca Muscholl (University Bordeaux)
- Catuscia Palamidessi (INRIA)
- Raimund Seidel (Saarland University and Schloss Dagstuhl – Leibniz-Zentrum für Informatik)
- Thomas Schwentick (TU Dortmund)
- Reinhard Wilhelm (Saarland University)

ISSN 1868-8969

http://www.dagstuhl.de/lipics

CONCUR 2017
Contents

Preface
Roland Meyer and Uwe Nestmann ............................................ 0.ix

Invited Papers
Forward Progress on GPU Concurrency
Alastair F. Donaldson, Jeroen Ketema, Tyler Sorensen, and John Wickerson .... 1:1–1:13
Admissibility in Games with Imperfect Information
Romain Brenguier, Arno Pauly, Jean-François Raskin, and Ocan Sankar ........ 2:1–2:23
Probabilistic Programming
Hongseok Yang ............................................................. 3:1–3:1
A New Notion of Compositionality for Concurrent Program Proofs
Azadeh Farzan and Zachary Kincaid ........................................ 4:1–4:11

Regular Papers
Bidirectional Nested Weighted Automata
Krishnendu Chatterjee, Thomas A. Henzinger, and Jan Otop ............... 5:1–5:16
k-Bounded Petri Net Synthesis from Modal Transition Systems
Uli Schlachter and Harro Wimmel ........................................ 6:1–6:15
A Characterisation of Open Bisimilarity using an Intuitionistic Modal Logic
Ki Yang Ahn, Ross Horne, and Alwen Tiu .................................. 7:1–7:17
Consistently-Detecting Monitors
Adrian Francalanza .......................................................... 8:1–8:19
Flow Logic
Orna Kupferman and Gal Vardi ............................................. 9:1–9:18
Rule Formats for Nominal Process Calculi
Luca Accoto, Ignacio Fábregas, Álvaro García-Pérez, Anna Ingólfsdóttir, and Yolanda Ortega-Mallén ..................................... 10:1–10:16
Divergence and Unique Solution of Equations
Adrien Durier, Daniel Hirschkoff, and Davide Sangiorgi ..................... 11:1–11:16
Controlling a Population
Nathalie Bertrand, Miheer Dewaskar, Blaise Genest, and Hugo Gimbert .... 12:1–12:16
The Robot Routing Problem for Collecting Aggregate Stochastic Rewards
Coverability Synthesis in Parametric Petri Nets
Nicolas David, Claude Jard, Didier Lime, and Olivier H. Roux ............ 14:1–14:16
Divide and Congruence III: Stability & Divergence
Wan Fokkink, Rob van Glabbeek, and Bas Luttik ........................................ 15:1–15:16

Checking Linearizability of Concurrent Priority Queues
Ahmed Bouajjani, Constantin Enea, and Chao Wang .............................. 16:1–16:16

Nash Equilibrium and Bisimulation Invariance
Julian Gutierrez, Paul Harrenstein, Giuseppe Perelli, and Michael Wooldridge ... 17:1–17:16

Goal-Driven Unfolding of Petri Nets
Thomas Chatain and Loïc Paulevé .................................................. 18:1–18:16

Probabilistic Automata of Bounded Ambiguity
Nathanaël Fijalkow, Cristian Riveros, and James Worrell .................... 19:1–19:14

Two Lower Bounds for BPA
Mingzhang Huang and Qiang Yin .................................................. 20:1–20:16

Infinite-Duration Bidding Games
Guy Avni, Thomas A. Henzinger, and Ventsislav Chonev ..................... 21:1–21:18

On the Power of Name-Passing Communication
Yuxi Fu ........................................................................ 22:1–22:15

The Power of Convex Algebras
Filippo Bonchi, Alexandra Silva, and Ana Sokolova ............................ 23:1–23:18

Refinement for Signal Flow Graphs
Filippo Bonchi, Joshua Holland, Dusko Pavlovic, and Paweł Sobociński .... 24:1–24:16

Brzozowski Goes Concurrent – A Kleene Theorem for Pomset Languages
Tobias Kappé, Paul Brunet, Bas Luttik, Alexandra Silva, and Fabio Zanasi ...... 25:1–25:16

Algebraic Laws for Weak Consistency
Andrea Cerone, Alexey Gotsman, and Hongseok Yang ....................... 26:1–26:18

Algorithms to Compute Probabilistic Bisimilarity Distances for Labelled Markov Chains
Qiyi Tang and Franck van Breugel ................................................. 27:1–27:16

On Decidability of Concurrent Kleene Algebra
Paul Brunet, Damien Pous, and Georg Struth .................................. 28:1–28:15

Model-Checking Counting Temporal Logics on Flat Structures
Normann Decker, Peter Habermehl, Martin Leucker, Arnaud Sangnier, and Daniel Thoma .................................................. 29:1–29:17

Concurrent Reversible Sessions
Ilaria Castellani, Mariangiola Dezani-Ciancaglini, and Paola Gianni ........ 30:1–30:17

Unbounded Product-Form Petri Nets
Patricia Bouyer, Serge Haddad, and Vincent Jugé .............................. 31:1–31:16

Efficient Coalgebraic Partition Refinement

The Complexity of Flat Freeze LTL
Benedikt Bollig, Karin Quaas, and Arnaud Sangnier ................................. 33:1–33:16
Higher-Order Linearisability
Andrzej S. Murawski and Nikos Tzevelekos ........................................ 34:1–34:18

Model Checking $\omega$-regular Properties for Quantum Markov Chains
Yuan Feng, Ernst Moritz Hahn, Andrea Turrini, and Shenggang Ying ........ 35:1–35:16

Uniform Sampling for Networks of Automata
Nicolas Basset, Jean Mairesse, and Michèle Soria ............................ 36:1–36:16

Tractability of Separation Logic with Inductive Definitions: Beyond Lists
Taolue Chen, Fu Song, and Zhilin Wu .............................................. 37:1–37:17

Data Multi-Pushdown Automata
Parosh Aziz Abdulla, C. Aiswarya, and Mohamed Faouzi Atig .......... 38:1–38:17

Towards an Efficient Tree Automata Based Technique for Timed Systems

On Petri Nets with Hierarchical Special Arcs
Preface

This volume contains the proceedings of the 28th Conference on Concurrency Theory, which was held in Berlin, Germany, on September 5–8, 2017. CONCUR 2017 was organized by Technische Universität Berlin and Technische Universität Braunschweig.

CONCUR is a forum for the development and dissemination of leading research in concurrency theory and its applications. The aim is to bring together researchers, developers, and students, exchange and discuss latest theoretical developments and learn about challenging practical problems. CONCUR is the reference annual event for researchers in the field.

The principal topics include basic models of concurrency such as abstract machines, domain-theoretic models, game-theoretic models, process algebras, graph transformation systems, Petri nets, hybrid systems, mobile and collaborative systems, probabilistic systems, real-time systems, biology-inspired systems, and synchronous systems; logics for concurrency such as modal logics, probabilistic and stochastic logics, temporal logics, and resource logics; verification and analysis techniques for concurrent systems such as abstract interpretation, atomicity checking, model checking, race detection, pre-order and equivalence checking, run-time verification, state-space exploration, static analysis, synthesis, testing, theorem proving, type systems, and security analysis; distributed algorithms and data structures: design, analysis, complexity, correctness, fault tolerance, reliability, availability, consistency, self-organization, self-stabilization, protocols. Also the theoretical foundations of more applied topics like architectures, execution environments, and software development for concurrent systems such as geo-replicated systems, communication networks, multiprocessor and multi-core architectures, shared and transactional memory, resource management and awareness, compilers and tools for concurrent programming, programming models such as component-based, object- and service-oriented can be found at CONCUR.

This edition of the conference attracted 86 full paper submissions. We would like to thank the authors for their interest in CONCUR 2017. After careful reviewing and discussions, the Program Committee selected 36 papers for presentation at the conference. Each submission was reviewed by at least three reviewers who wrote detailed evaluations and gave insightful comments. The Conference Chairs warmly thank the members of the Program Committee and the additional reviewers for their excellent work, as well as for the constructive discussions. The full list of reviewers is available as part of these proceedings.

The conference program was greatly enriched by the invited talks by Hongseok Yang (University of Oxford, UK), Azadeh Farzan (University of Toronto, Canada), Madan Musuvathi (Microsoft Research, USA), and Jean-Francois Raskin (Université libre de Bruxelles, Belgium). Moreover, Alastair Donaldson (Imperial College London, UK), Pawel Sobocinski (University of Southampton, UK), and Viktor Vafeiadis (Max Planck Institute for Software Systems, Germany) kindly agreed to contribute tutorials. The invited and the tutorial talks cover a broad range of topics from traditional concurrency theory and domains through games to the analysis of GPU kernels and the semantics of C++ in the presence of concurrency. The abstracts and invited papers are available as part of these proceedings. We thank the speakers for having accepted our invitation.

This year, the conference was jointly organized with the 14th International Conference on Quantitative Evaluation of SysTems (QEST), the 15th International Conference on Formal Modelling and Analysis of Timed Systems (FORMATS), and the 14th European Performance Engineering Workshop (EPEW) in an overarching event QONFEST. QONFEST included five more satellite events, the combined 24rd International Workshop on Expressiveness in

28th International Conference on Concurrency Theory (CONCUR 2017).
Editors: Roland Meyer and Uwe Nestmann
Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany
Preface

Concurrency and 14th Workshop on Structural Operational Semantics (EXPRESS/SOS), the 1st Workshop on Recent Advances in Concurrency and Logic (RADICAL), the 7th Young Researchers Workshop on Concurrency Theory (YR-CONCUR), the 6th IFIP WG 1.8 Workshop on Trends in Concurrency Theory (TRENDS), and the 9th International Workshop on Practical Applications of Stochastic Modelling (PASM).

The CONCUR proceedings are available for open access via LIPIcs, and we thank the Marc Herbstritt and Sebastian Schweizer for helping us with the preparation.

Last, but not least, we thank the authors and the participants for making this year’s CONCUR a successful event.

Roland Meyer (TU Braunschweig)
Uwe Nestmann (TU Berlin)
Committees

Program Committee
Jade Alglave
Mohamed Faouzi Atig
Paolo Baldan
Johannes Borgström
Luis Caires
Pedro R. D’Argenio
Josée Desharnais
Constantin Enea
Javier Esparza
Wan Fokkink
Rob van Glabbeek
Stefan Göller
Thomas Hildebrandt
Naoki Kobayashi
Antonín Kučera
Jérôme Leroux
Roland Meyer
K Narayan Kumar
Uwe Nestmann
Catuscia Palamidessi
Alexander Rabinovich
Davide Sangiorgi
Pawel Sobocinski
Vasco Thudicum Vasconcelos
Walter Vogler
Tomáš Vojnar
Igor Walukiewicz
Heike Wehrheim
Josef Widder
Thomas Wies

Co-Chairs
Gianluigi Zavattaro
Lijun Zhang

Steering Committee
Jos Baeten
Pedro R. D’Argenio
Javier Esparza
Joost-Pieter Katoen
Kim Guldstrand Larsen
Ugo Montanari
Catuscia Palamidessi

Organizing Committee
Uwe Nestmann, TU Berlin
(KONFEST General Co-Chair)
Katinka Wolter, FU Berlin
(KONFEST General Co-Chair)
Kirstin Peters, TU Berlin
(CONCUR Workshop Chair)
External Reviewers

Samy Abbes
Antonis Achilleos
C. Aiswarya
S. Akshay
Robert Atkey
Giorgio Bacci
Luca Bernardinello
Nathalie Bertrand
Michael Blondin
Benedikt Bollig
Tomas Brazdil
Yuliya Butkova
Marco Carbone
Raffaella Carbone
Sofia Cassel
Andrea Cerone
Yu-Fang Chen
Peter Chini
Dmitry Chistikov
Corina Cirstea
Lorenzo Clemente
Emanuele D’Osualdo
Søren Debois
Giorgio Delzanno
Romain Demangeon
Yuxin Deng
Brijesh Dongol
Adrien Durier
Christian Eisenraut
Jerome Feret
Jan Fiedor
Adrian Francalanza
Hongfei Fu
Paul Gastin
Marie-Claude Gaudel
Daniel Gburek
Dan R. Ghica
Daniele Gorla
Nicolas Guenot
Peter Habermehl
Christopher Hampson
Rolf Hennicker
Frédéric Herbreteau
Tom Hirschowitz
Lukas Holik
Radu Iosif
Stefan Jaax
Swen Jacobs
Marie-Christine Jakobs
Petr Jancar
Peter Jipsen
Edon Kelmendi
Henning Kerstan
Sophia Knight
Jan Kretinsky
Siddharth Krishna
Denis Kuperberg
Jürgen König
Marijana Lazic
Ondrej Lengal
Yong Li
Sylvain Lombardy
External Reviewers

Robert Lorenz
Michele Loreti
Gerald Luettgen
Konstantinos Mamouras
Radu Mardare
Andrea Marin
Nicolas Markey
Paulo Mateus
Richard Mayr
Claudio Antares Mezzina
Samuel Mimram
Benjamin Monmege
Andrzej Murawski
Sebastian Musella
Bernhard Möller
Elisabeth Neumann
Petr Novotný
Jan Obdrzalek
Joachim Parrow
Kirstin Peters
Gustavo Petri
M. Praveen
Gabriele Puppis
Tahiry Rabehaja
Ahmed Rezine
Christina Rickmann
Adam Rogalewicz
Jurriaan Rot
David Šafránek
Prakash Saivasan
Arnaud Sangnier
Ocan Sankur
Ocan Sankur
Zdenek Sawa

Alceste Scalas
Carsten Schuermann
Henning Seidler
Salomon Sickert
Michal Skrzypczak
Ana Sokolova
Marco Solieri
Sadegh Soudjani
Sam Staton
Ilina Stoilkovska
K V Subrahmaniam
Kohei Suenaga
Grégoire Sutre
Géraud Sénizergues
Daniel Thoma
Simone Tini
Takeshi Tsukada
Andrea Turrini
Valeria Vignudelli
Mahesh Viswanathan
Christoph Wagner
Herbert Wiklicky
Sebastian Wolff
Damien Zufferey
List of Authors

Parosh Aziz Abdulla  
Uppsala University  
Sweden  
parosh@it.uu.se

Luca Aceto  
Reykjavik University  
Iceland  
luca@ru.is

Ki Yung Ahn  
Nanyang Technological University  
Singapore  
yaki@ntu.edu.sg

C. Aiswarya  
Chennai Mathematical Institute  
India  
aiswaryanitc@gmail.com

S. Akshay  
Indian Institute of Technology Bombay  
India  
akshayss@cse.iitb.ac.in

Mohamed Faouzi Atig  
Uppsala University  
Sweden  
mohamed_faouzi.atig@it.uu.se

Guy Avni  
IST Austria  
Austria  
guy. avi@ist.ac.at

Nicolas Basset  
Université libre de Bruxelles  
Belgium  
nicolas.basset@ulb.ac.be

Nathalie Bertrand  
IRISA  
France  
nathalie.bertrand@inria.fr

Benedikt Bollig  	CNRS, ENS Paris-Saclay  	France  	bollig@lsv.fr

Filippo Bonchi  	CNRS, ENS-Lyon  	France  	filippo.bonchi@ens-lyon.fr

Ahmed Bouajjani  	IRIF, University Paris Diderot  	France  	abou@irif.fr

Patricia Bouyer  	Université Paris-Saclay  	France  	bouyer@lsv.fr

Romain Brenguier  	Oxford University  
United Kingdom  
romain.brenguier@gmail.com

Franck van Breugel  	York University, Toronto  
Canada  
franck@ee.cs.yorku.ca

Paul Brunet  	ENS de Lyon, France  
University College London, UK  
paul@brunet-zamansky.fr

Ilaria Castellani  	Université Côte d’Azur, INRIA  
France  
ilaria.castellani@inria.fr

Andrea Cerone  	Imperial College London  
United Kingdom  
a.cerone@imperial.ac.uk

Supratik Chakraborty  	Indian Institute of Technology Bombay  
India  
akshayss@gmail.com

Thomas Chatain  	École Normale Supérieure de Cachan  
France  
thomas.chatain@lsv.ens-cachan.fr
Krishnendu Chatterjee
IST Austria
Austria
krish.chat@ist.ac.at

Taolue Chen
Birkbeck, University of London
United Kingdom
taolue.chen@gmail.com

Ventsislav Chonev
Max Planck Institute for Software Systems
Germany
vencho@mpi-sws.org

Ankush Das
Carnegie Mellon University
USA
ankushd@cs.cmu.edu

Nicolas David
University of Nantes
France
nicolas.david1@univ-nantes.fr

Normann Decker
Universität zu Lübeck
Germany
decker@isp.uni-luebeck.de

Miheer Dewaskar
University of North Carolina
USA
miheer.dew@gmail.com

Mariangiola Dezani-Ciancaglini
Università di Torino
Italy
dezani@di.unito.it

Rayna Dimitrova
Max Planck Institute for Software Systems
Germany
rayna@mpi-sws.org

Alastair Donaldson
Imperial College London
United Kingdom
alastair.donaldson@imperial.ac.uk

Ulrich Dorsch
Friedrich-Alexander-Universität
Erlangen-Nürnberg
Germany
ulrich.dorsch@fau.de

Adrien Durier
École Normale Supérieure de Lyon,
Università di Bologna
France
adrien.durier@gmail.com

Constantin Enea
IRIF, University Paris Diderot
France
cenea@irif.fr

Ignacio Fábregas
IMDEA Software Institute, Madrid
Spain
fabregas@ucm.es

Azadeh Farzan
University of Toronto
Canada
azadeh@cs.toronto.edu

Yuan Feng
University of Technology Sydney
Australia
yuan.feng@uts.edu.au

Nathanaël Fijalkow
The Alan Turing Institute of Data Science,
London
United Kingdom
nfijalkow@turing.ac.uk

Wan Fokkink
Vrije Universiteit Amsterdam
The Netherlands
w.j.fokkink@vu.nl

Adrian Francalanza
University of Malta
Malta
adrian.francalanza@um.edu.mt

Yuxi Fu
Shanghai Jiao Tong University
China
fu-yx@cs.sjtu.edu.cn
Authors

Vincent Jugé
Université Paris-Saclay
France
vincent.juge@polytechnique.edu

Tobias Kappé
University College London
United Kingdom
tkappe@cs.ucl.ac.uk

Jeroen Ketema
Embedded Systems Innovation by TNO
Netherlands
jeroen.ketema@tno.nl

Zachary Kincaid
Princeton University
USA
zkincaid@cs.princeton.edu

Shankara Narayanan Krishna
IIT Bombay
India
krishnas@cse.iitb.ac.in

Orna Kupferman
The Hebrew University
Israel
orna@cs.huji.ac.il

Martin Leucker
Universität zu Lübeck
Germany
leucker@isp.uni-luebeck.de

Didier Lime
École Centrale de Nantes
France
didier.lime@ec-nantes.fr

Bas Luttik
Eindhoven University of Technology
The Netherlands
s.p.luttik@tue.nl

Jean Mairesse
CNRS - Sorbonne Universités
France
jean.mairesse@lip6.fr

Rupak Majumdar
Max Planck Institute for Software Systems
Germany
rupak@mpi-sws.org

Stefan Milius
Friedrich-Alexander-Universität Erlangen-Nürnberg
Germany
stefan.milius@fau.de

Andrzej S. Murawski
University of Warwick, Coventry
United Kingdom
A.Murawski@warwick.ac.uk

Yolanda Ortega-Mallén
Universidad Complutense de Madrid
Spain
yolanda@ucm.es

Jan Otop
University of Wroclaw
Poland
jotop@cs.uni.wroc.pl

Loïc Paulevé
CNRS
France
loic.pauleve@lri.fr

Arno Pauly
Université Libre de Bruxelles, U.L.B.
Belgium
arno.m.pauly@gmail.com

Dusko Pavlovic
University of Hawaii at Manoa
Hawaii
dusko@hawaii.edu

Giuseppe Perelli
University of Oxford
United Kingdom
giuseppe.perelli@cs.ox.ac.uk

Damien Pous
ENS de Lyon
France
damien.pous@ens-lyon.fr

Vinayak Prabhu
Max Planck Institute for Software Systems
Germany
vinayak@mpi-sws.org
Authors

Karin Quaas
Universität Leipzig
Germany
quaas@informatik.uni-leipzig.de

Jean-François Raskin
Université Libre de Bruxelles, U.L.B.
Belgium
jraskin@ulb.ac.be

Cristian Riveros
Pontificia Universidad Católica de Chile
Chile
cristian.riveros@uc.cl

Olivier H. Roux
École Centrale de Nantes
France
olivier-h.roux@ec-nantes.fr

Sai Sandeep
Indian Institute of Technology Bombay
India
saisandeep@cse.iitb.ac.in

Davide Sangiorgi
Università di Bologna
Italy
davide.sangiorgi@gmail.com

Arnaud Sangnier
IRIF, Univ Paris Diderot
France
sangnier@irif.fr

Ocan Sankur
CNRS - IRISA Rennes
France
ocan.sankur@irisa.fr

Ilias Sarkar
IIT Bombay
India
ilias@cse.iitb.ac.in

Uli Schlachter
Carl von Ossietzky Universität, Oldenburg
Germany
uli.schlachter@informatik.uni-oldenburg.de

Lutz Schröder
Friedrich-Alexander-Universität
Erlangen-Nürnberg
Germany
lutz.schroeder@fau.de

Alexandra Silva
University College London
United Kingdom
alexandra.silva@ucl.ac.uk

Pawel Sobocinski
University of Southampton
United Kingdom
ps@ecs.soton.ac.uk

Ana Sokolova
University of Salzburg
Austria
ana.sokolova@cs.uni-salzburg.at

Fu Song
ShanghaiTech University
China
songfu@shanghaitech.edu.cn

Tyler Sorensen
Imperial College London
United Kingdom
tyler.sorensen15@imperial.ac.uk

Michèle Soria
UPMC - Sorbonne Universités
France
michelle.soria@lip6.fr

Sadegh Soudjani
Max Planck Institute for Software Systems
Germany
sadegh@mpi-sws.org

Georg Struth
The University of Sheffield
United Kingdom
g_struth@sheffield.ac.uk

Qiyi Tang
York University, Toronto
Canada
qiyitang@eecs.yorku.ca
Authors

Daniel Thoma
Universität zu Lübeck
Germany
thoma@isp.uni-luebeck.de

Alwen Tiu
Nanyang Technological University
Singapore
atiu@ntu.edu.sg

Andrea Turrini
Institute of Software, Chinese Academy of Sciences, Beijing
China
turrini@ios.ac.cn

Nikos Tzevelekos
Queen Mary University of London
United Kingdom
nikos.tzevelekos@qmul.ac.uk

Gal Vardi
The Hebrew University
Israel
gal.vardi@mail.huji.ac.il

Chao Wang
IRIF, University Paris Diderot
France
wangch@irif.fr

James Worrell
University of Oxford
United Kingdom
james.worrell@cs.ox.ac.uk

Zhilin Wu
Chinese Academy of Sciences
China
wuzl@ios.ac.cn

Hongseok Yang
University of Oxford
United Kingdom
hongseok.yang@cs.ox.ac.uk

Qiang Yin
Beihang University
China
yinqiang@buaa.edu.cn

Shenggang Ying
University of Technology Sydney
Australia
shenggang.ying@uts.edu.au

Fabio Zanasi
University College London
United Kingdom
f.zanasi@ucl.ac.uk
Forward Progress on GPU Concurrency

Alastair F. Donaldson¹, Jeroen Ketema², Tyler Sorensen³, and John Wickerson⁴

¹ Department of Computing, Imperial College London, UK
   alastair.donaldson@imperial.ac.uk
² Embedded Systems Innovation by TNO, Eindhoven, the Netherlands
   jeroen.ketema@tno.nl
³ Department of Computing, Imperial College London, UK
   tyler.sorensen15@imperial.ac.uk
⁴ Department of Electrical and Electronic Engineering, Imperial College London, UK
   j.wickerson@imperial.ac.uk

Abstract

The tutorial at CONCUR will provide a practical overview of work undertaken over the last six years in the Multicore Programming Group at Imperial College London, and with collaborators internationally, related to understanding and reasoning about concurrency in software designed for acceleration on GPUs. In this article we provide an overview of this work, which includes contributions to data race analysis, compiler testing, memory model understanding and formalisation, and most recently efforts to enable portable GPU implementations of algorithms that require forward progress guarantees.

1998 ACM Subject Classification D.1.3 Concurrent Programming

Keywords and phrases GPUs, concurrency, formal verification, memory models, data races

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2017.1

Category Invited Talk

1 Introduction

Graphics processing units (GPUs) offer a large degree of parallelism at a relatively low cost, and are now routinely applied to the acceleration of a wide variety of computational tasks that go well beyond the domain of graphics (see e.g. [39] for details of many application areas and developments).

It is invariably harder to design a software application that takes advantage of GPU parallelism than it is to write a sequential version of the application that runs only on the CPU. Furthermore, parallel programming for GPUs is in many ways more complicated than parallel programming for multi-core CPUs. This is because achieving high performance requires working in low level languages, such as CUDA [33] and OpenCL [24], which provide close-to-the-metal language features to enable mapping an algorithm to the architectural capabilities of a device. Numerous high level programming models have been proposed to ease the burden of GPU programming, via automatic generation of low level code, but as yet are not widely adopted.

Low level programming of GPUs is made challenging by traditional concurrency bugs such as data races, by issues related to relaxed memory, and by the constrained hardware execution model on which software executes. Furthermore, reliable production compilers for
GPUs are hard to maintain due to the challenge of keeping pace with changing architectures and evolving source languages, and because GPU-specific optimisations—e.g. ones that aim to reduce control-flow divergence between threads—are important for performance but error-prone to implement.

After giving a brief overview of the GPU programming model (Section 2), we present an overview of a line of work pursued by the Multicore Programming Group at Imperial College London, with various collaborators, which has contributed to: automated data race analysis (Section 3), compiler testing (Section 4), memory models (Section 5), and forward progress guarantees to support blocking algorithms (Section 6). While we principally focus on the approach taken by our line of work, we briefly discuss related approaches. In each area, we also touch on open problems that will require future research to solve.

2 Overview of the GPU Programming Model

GPUs are programmed using an SPMD (single program, multiple data) model, in which a large number of processing elements (PEs) all execute the same program, with each PE operating on a different subset of some shared data. At the hardware level, PEs are typically grouped into multiprocessors (see Figure 1). Each PE generally has access to a few hundred bytes of private memory; all the PEs in the same multiprocessor have a few kilobytes of shared memory available; and a large amount of global memory is shared between all multiprocessors.

The two main programming languages for writing SPMD programs are OpenCL [24] and CUDA [33], both of which derive from the C programming language. The languages follow the above sketched hardware hierarchy quite closely by subdividing the execution of a program among a number of work-items (or threads)—mapping to PEs. These work-items are grouped into (multi-dimensional) work-groups (or thread-blocks)—mapping to multiprocessors—which in turn are grouped into (multi-dimensional) NDRanges (or grids). A programmer specifies the program to be executed by a single work-item—called a GPU kernel—and defines the work-group and NDRange sizes that should be used for execution. The OpenCL programming model uses the terms work-items, work-groups, and NDRanges, while CUDA uses threads, thread-blocks, and grids. For the remainder of the article we use the OpenCL terminology, except that we find it more natural to use thread rather than work-item.

An OpenCL kernel computing a prefix-sum (or scan) [25] is presented in Figure 2. The kernel is intended to be executed by a single 1-dimensional work-group. The keyword global indicates that the arrays in and out reside in global memory. Similarly, the keyword local (not used in the example) indicates that an array resides in shared memory. Any variable declared without the global or local keyword is thread-private. To enable each thread to
operate on different data, several functions are provided that allow a thread to access its unique id. One of these functions is get_local_id, which yields the unique id of a thread within a work-group.

Except when performing the most basic computations, threads typically need to share intermediate results. In OpenCL and CUDA, these intermediate results are traditionally shared with the help of synchronisation barriers: each thread writes its intermediate results to local or global memory and then calls the barrier function. The barrier function stalls the thread until all other threads within the same work-group have also called the function and all outstanding memory operations have been committed. The argument of the barrier function indicates whether operations on local or global memory need to be committed. In the case of the GPU kernel of Figure 2, only operations on global memory are committed.

As stressed above, barriers can only be used for communication within work-groups; they cannot be used for communication between work-groups. The same holds for early implementations of atomic operations on GPUs. This was rectified with OpenCL 2.0, which defines atomic operations that can operate between work-groups. The semantics of OpenCL 2.0 atomics are subtle, and expressing them precisely is a research endeavour we are pursuing, thus we postpone a more detailed discussion of them to Section 5. Finally, it is worth mentioning that the new NVIDIA CUDA 9.0 provides primitives for barrier synchronisation across work-groups [34].

3 Static Data Race Analysis

Data races are an important kind of defect that affect shared memory concurrent programs, and GPU kernels in particular. Informally, a GPU kernel exhibits a data race if it is possible for two distinct threads, $t_1$ and $t_2$, to issue memory operations accessing a common location $m$ such that: at least one of the operations is non-atomic, at least one of the operations modifies $m$, and $t_1$ and $t_2$ do not synchronise between the operations. The semantics of data races are not well defined in CUDA nor in older versions of OpenCL, and data races are a form of undefined behaviour in OpenCL 2.0 [24] and later. As well as being defects in their own right, data races can lead to unwanted nondeterminism, which can cause other bugs that are hard to reproduce and fix.

To aid programmers in reasoning about data races in GPU kernels we spent several years designing GPUVerify, a static data race analysis tool [11, 10, 4]. GPUVerify caters primarily for the traditional style of GPU programming where threads in distinct work-groups do not
communicate during the lifetime of a kernel, and where intra-work-group communication is exclusively via synchronisation barriers.

The traditional setting affords a key property: when a GPU kernel is executed on a given input, if any thread schedule can exhibit a data race, then all thread schedules exhibit a data race. We briefly sketch the argument for this. Let us call a data race occurring in a given thread schedule a principal race if the associated memory accesses \( a_1 \) and \( a_2 \) are guaranteed to occur, regardless of the instructions executed by other threads. That is, the conditions for the associated threads to issue \( a_1 \) and \( a_2 \) arise directly from the input on which the GPU kernel is executed, and do not depend on the thread schedule. Clearly, if a thread schedule exhibits a principal race involving accesses \( a_1 \) and \( a_2 \), then every thread schedule must exhibit this principal race. Furthermore, in the traditional GPU programming model, every thread schedule that exhibits a race is guaranteed to exhibit a principal race. This is because a traditional GPU kernel behaves entirely deterministically in the absence of a data race [16].

Threads in different work-groups have no means of synchronising, and thus can only influence one another by racing. Threads in the same work-group can only synchronise via barriers. Hence, in the absence of data races, the execution of each thread is independent of the actions of all other threads until barrier synchronisation occurs, and the state of each thread upon reaching a barrier is independent of the chosen thread interleaving. Repeating this argument for each following barrier, we see that traditional GPU kernel execution is deterministic. Because a race-free GPU kernel is deterministic, a kernel exhibiting a race is guaranteed to have a fixed set of principal races and, hence, if any schedule exhibits a race then all do.

GPUVerify leverages the above result to reduce the problem of reasoning about data races in highly parallel GPU kernels to reasoning about assertions in a sequential program. This is achieved by translating a parallel kernel into a sequential program that models one particular thread schedule between each pair of barrier synchronisation points in the kernel, tracking the reads and writes of threads, and using assertions to determine the conditions under which data races occur. In practice, GPUVerify uses a schedule in which threads execute in lock-step [18].

Because GPU kernels are often executed by hundreds or even thousands of threads, it would not be practical to model all these threads explicitly. Instead, GPUVerify exploits the fact that data races occur pairwise, and only models execution of a kernel by a pair of threads. The identities of these threads are made symbolic in the sequential program that GPUVerify generates, so that reasoning is in fact performed over all possible pairs. To over-approximate the effects of additional threads, the shared state of the GPU kernel is made abstract. By default GPUVerify uses an extremely coarse abstraction, where other threads are assumed to have arbitrary effects on the shared state [10].

The sequential program generated by GPUVerify is encoded in the Boogie intermediate verification language [5], whose supporting tool—also called Boogie—generates verification conditions that are discharged by SMT solvers such as Z3 [19] and CVC4 [6]. To handle loops, Boogie requires invariants to be supplied. For this purpose we have equipped GPUVerify with a tailored engine for loop invariant inference, which is by now relatively mature. The engine works by speculating candidate loop invariants via a set of rules, which we devised through careful study of a large set of GPU kernels [9]. The candidate invariants speculated for a kernel are fed to the Houdini algorithm [20], which computes the largest conjunction of

---

1 We assume here that concurrency is the only possible cause of nondeterminism; sources of nondeterminism due to e.g. accessing invalid memory locations are an orthogonal concern.
candidates that form an inductive invariant for the kernel. The computed invariant is then used in an attempt to prove freedom from data races.

Several other methods for testing and verifying properties of GPU kernels have been proposed. These include approaches based on dynamic analysis [35, 48, 28], verification via SMT solving [29, 38, 44], symbolic execution [30, 17], and program logic [12, 26].

Open problems

Despite operating reasonably automatically on a relatively large set of GPU kernels [4, 9], the GPUVerify technique has three main limitations.

First, the loop invariant inference method used by the tool can be inefficient, due to the sheer number of candidate invariants that are speculated. Many of the candidates turn out to be incorrect or non-inductive, so that many iterations of the Houdini algorithm are required to refute them, each iteration requiring one or more expensive calls to an SMT solver. We believe that a significantly more efficient mechanism for computing relevant loop invariants could be built by performing abstract interpretation over carefully-designed abstract domains that capture the memory access patterns of GPU kernels.

Second, the shared state abstraction employed by GPUVerify leads to false alarms for kernels whose race-freedom depends on richer properties of the shared state. To aid in the verification of such kernels, GPUVerify supports barrier invariants—properties of the shared state that are proven to be satisfied by threads individually on entry to a barrier synchronisation operation, and which can thus be assumed to hold for all threads on exit from the barrier [15]. However, barrier invariants must currently be specified manually, and have only been investigated practically in one domain so far: reasoning about parallel prefix sums. Relatedly, our recent work on termination analysis for GPU kernels shows that the coarse shared state abstraction used by GPUVerify often suffices for proving kernel termination in a thread-modular manner, but identifies a number of cases where richer shared state abstractions are required [23].

Third, GPUVerify fundamentally exploits the traditional GPU programming model where barriers are the only means for synchronisation. As discussed further in Sections 5 and 6, modern GPU kernels increasingly go beyond the boundaries of this simple computational model, using atomic operations to implement fine-grained concurrent algorithms. In general, reasoning about data race-freedom of GPU kernels that use atomic operations is just as hard as reasoning about data race-freedom for arbitrary concurrent programs. In particular, a race-free GPU kernel need not behave deterministically if it uses atomic operations. This breaks the property that allows GPUVerify to reason about a kernel via translation to a sequential program. Although GPUVerify can handle some limited use cases for atomic operations [7], the tool over-approximates more general uses of atomics, so that it will report false alarm data races for algorithms that, for instance, protect shared data structures using mutexes built via atomic operations. Nevertheless, we conjecture that the contexts in which GPU programmers use atomic operations in practice are likely to be limited and idiomatic, and that there may be scope for targeted analyses that exploit this idiomatic nature.

In related work, we note that GKLEE, another GPU race detection tool [30], has also been extended to reason about atomics in some scenarios [14], and that reasoning about atomic operations in GPU kernels via resource invariants in separation logic has recently been proposed [3].
The race analysis provided by GPUVerify (see Section 3) operates on the intermediate representation of the LLVM compiler framework, generated by Clang’s OpenCL and CUDA front-ends. More generally, it is common for static race analysis tools to operate at the level of program source code, or at some intermediate representation associated with a particular tool chain. Either way, an analysis that establishes properties of a program at some higher-than-binary level of abstraction relies on correct downstream compilation.

Aware that guarantees provided by GPUVerify are conditional on the reliability of CUDA and OpenCL compilers, we undertook research into testing such compilers. We focused on testing OpenCL compilers using two methods: random differential testing [32], where compilers are cross-checked against each other using randomly-generated programs, and equivalence modulo inputs testing [27], where a single compiler is tested via a family of programs that, for a given input, ought to yield equivalent results. In both cases, mismatches are indicative of compiler bugs. Both methods are examples of fuzzing, where a system is tested against randomly-generated inputs.

We built a tool, CLsmith [31], which extends the Csmith generator for C programs [47] to the domain of OpenCL. It was relatively simple to extend Csmith to generate “embarrassingly parallel” OpenCL kernels in which threads do not communicate at all. More challenging was to devise methods for generating OpenCL kernels in which threads do communicate—either using barriers or atomic operations—but in a manner such that the generated kernels are guaranteed to be free from data races and to compute deterministic results.

We armed CLsmith with three modes for generating such kernels: barrier mode, where generated kernels are equipped with shared arrays, indexed in a manner that avoids data races; atomic-sections mode, where atomic operations are used to build sections of code that can only be executed by a single thread, structured such that the particular thread that executes a section depends on the thread schedule, but such that the side-effects associated with executing the section are independent of the specific thread that executes the section; and atomic-reductions mode, where associative, commutative atomic operations (such as atomic_add and atomic_min) are used to perform reductions on data values computed locally by individual threads, ultimately leading to deterministic results.

Via an experimental campaign applying CLsmith to 21 OpenCL (device, compiler)-configurations, covering a range of GPU, CPU, FPGA, and emulator implementations [31], we discovered more than 50 OpenCL compiler bugs, most affecting commercial implementations. Surprisingly, and disappointingly from an academic perspective, these bugs were almost exclusively sequential: basic compilation bugs that could manifest in a single-threaded kernel. We found a large number of defects affecting programs that manipulate data via user-defined structures. While we found a small number of bugs that required barrier synchronisation operations to be present in order to manifest, even these did not appear to be directly concurrency-related: they could affect kernels that, despite featuring barriers, did not depend on inter-thread communication for result computation.

As argued in Section 3, kernels that only use barrier operations for inter-thread communication are guaranteed to be deterministic if they are race-free. However, atomic operations open the possibility for race-free kernels to be nondeterministic.
Open problems

The fact that our testing methods did not identify any compiler bugs that were directly concurrency-related could be because (a) OpenCL compilers do not yet perform sophisticated concurrency-related optimisations, (b) our method did in fact trigger concurrency-related compiler bugs, but in addition triggered so many basic sequential compiler bugs that the effects of the concurrency-related bugs were masked, or (c) the methods we investigated so far for detecting concurrency-related bugs are too simple.

Possibility (a) could be investigated with reference to open source compilers, such as the Intel back-end for the Beignet open source implementation of OpenCL. Running a similar testing campaign against more mature OpenCL compilers in the future might shed light on possibility (b).

As we discuss further in Section 5, OpenCL 2.0 includes a full suite of well-defined atomic operations, together with a memory model specification. With respect to possibility (c), this raises the problem of how to generate random OpenCL kernels that exercise these concurrency-related features in non-trivial yet predictable ways, to test more thoroughly the compilation of concurrency primitives.

5 GPU Memory Models

As mentioned in Section 3, many GPU programmers have been going beyond the traditional barrier synchronous model of computation, writing fine-grained concurrent algorithms that manipulate irregular data structures, e.g. graphs. This raises the question of what sort of memory models GPU architectures present, and in particular, what values a thread might observe when reading from a shared memory location.

Empirically, we used litmus tests to investigate the memory models of several NVIDIA and AMD GPUs [1], showing (a) that they do exhibit relaxed (i.e. non-sequentially consistent) behaviour, and (b) that several programming idioms relied on by high level algorithms assume that such relaxed behaviour cannot occur, and thus are not guaranteed to work correctly on these platforms. In follow-up work we substantiated this possibility by designing a testing framework geared towards automatically provoking relaxed memory bugs in GPU-accelerated applications [40].

In large part, the issues that we discovered in these works stem from the lack of a memory model specification in CUDA, and in OpenCL prior to OpenCL 2.0. The OpenCL 2.0 specification [24] provides a memory model specification based on the C11 memory model [22]. However, the memory model is complicated by the fact that OpenCL features a hierarchical organisation of threads and memory spaces, as discussed in Section 2 and illustrated by Figure 1. For example, threads in the same work-group can use fast shared memory to communicate, while threads in different work-groups must communicate via slower global memory. OpenCL also features shared virtual memory (not depicted in the overview of Figure 1), through which device threads can communicate with threads running on the host processor. The situation is further complicated by the fact that memory accesses are scoped. For example, a write to global memory can have device scope, so that its effects are visible to all threads running on a device, or the more restricted work-group scope, so that although the write will eventually propagate to all threads, the write can only contribute to reliable synchronisation with threads in the same work-group. These subtleties

3 https://www.freedesktop.org/wiki/Software/Beignet/
are intended to provide optimisation opportunities for OpenCL programmers and language implementers. Our attempt to formalise the memory model [8] led to the identification of some fundamental flaws in the way the model handles the class of sequentially-consistent (SC) memory accesses. SC is the default mode for accessing memory, and is intended to be the easiest for programmers to reason about (but not the most efficient), because all SC accesses in a program are guaranteed to be executed in some total order on which all threads agree. Our formalisation exposed a tension between this total order (which is visible to all threads), and OpenCL’s scoping mechanism (which is intended to restrict visibility to just some threads). We were able to construct a small OpenCL program that involves SC accesses on two different devices, and show that even if these accesses are scoped to be visible only within the device that performs them, the compiler is still obliged to enforce an inter-device order between them [8, Example 10]. As such, the existing OpenCL memory model provides guarantees that are too strong to be efficiently implemented by a compiler. In our work, we suggest how the situation could be resolved in a future revision of the OpenCL memory model by enforcing a total order only between SC memory accesses that have the same scope [8, §5].

Although OpenCL’s scoping mechanism allows some synchronisation patterns to be optimised, its usefulness is hampered by its requirement that two memory accesses can only synchronise if both use a scope that is wide enough to encompass the other. This symmetry inhibits the popular work-stealing pattern [13], which is often used to implement algorithms with irregular workloads. In order for a work-group to allow its work to be stolen by another work-group, it must scope all of its memory accesses widely enough to encompass all the work-groups that might wish to steal. Using such a wide scope is unnecessary in the common case when no stealing occurs and visibility is only needed within the current work-group. To rectify this shortcoming, Orr et al. have proposed remote-scope promotion (RSP) [36]. In their proposal, memory accesses can default to the intra-work-group scope (for good performance in the common case), and when a thread wishes to steal from another work-group, it invokes RSP to temporarily widen the scope of the victim’s memory accesses, so that it can perform the necessary synchronisation to steal work.

Using the Isabelle proof assistant and the Herd memory model simulator [2], we formalised Orr et al.’s RSP extension and their proposed mapping to AMD’s prototype GPU architecture [45]. These efforts brought to light several corner cases in the design, which we discussed in detail with Orr et al. We also discovered two serious flaws in the proposed mapping, both of which could lead to incorrect results being calculated. We were able to propose fixes in both cases. This work emphasises the value of applying formal methods to the early-stage design of hardware support for concurrent programming language features.

In more recent work, we have gone on to develop techniques for making the bug-finding process applied to Orr et al.’s work more automatic. Our Memalloy tool [46] takes any source language memory model, any target architecture memory model, and a description of a compilation mapping from source language to target architecture, and uses automatic constraint-solving technology to seek programs with behaviours that are forbidden at the source language level but are nonetheless observable on the target architecture when the compilation mapping is applied. For instance, the tool is able to reproduce (and minimise) our two manually-found RSP-related bugs in under two hours. We also used Memalloy to explore the compilation process from OpenCL to NVIDIA GPUs, and thereby discovered that the memory model we had deduced for NVIDIA GPUs through litmus testing (see the beginning of this section) was too weak to support the natural compilation scheme for OpenCL. After strengthening (and re-validating) the NVIDIA model, Memalloy was able to prove (up to a bound) the absence of memory-related bugs in the compilation scheme.
Open problems

There remains work to be done to investigate how the OpenCL memory model is implemented on the other kinds of devices that are OpenCL targets, such as FPGAs and digital signal processors (DSPs). For instance, Intel’s OpenCL compiler for its FPGAs supports some of OpenCL’s atomic operations, and it may be possible to employ formal methods to help ascertain whether they are implemented correctly and as efficiently as possible.

Also, existing studies of GPU memory models (both at the language level and the architecture level) have focused primarily on the single-device setting. However, OpenCL is designed to allow multiple devices to communicate with each other and with the host processor (via shared virtual memory). There is a need for techniques to help ascertain whether the OpenCL memory model has been implemented correctly and efficiently in this case. Furthermore, when shared virtual memory is used in ‘fine-grained system’ mode (whereby devices have simultaneous access to the entirety of the host’s memory), there is a need to understand how the OpenCL memory model (as implemented on those devices) interacts with the memory model of the host processor. A general theory of how to compose memory models may be called for.

6 Blocking Algorithms and Forward Progress

Many parallel programs on traditional multicore systems (e.g. CPUs) have blocking behaviours. That is, one thread in the system will spin, waiting for another thread in the system to reach a certain point in the program. These interactions are typically orchestrated using flag values in a shared memory region. Common examples of concurrent idioms with blocking behaviours are mutexes and synchronisation barriers. In order to execute correctly (e.g. terminate or make progress), blocking behaviours require certain properties from the thread scheduler. Namely, if a thread $t_1$ is waiting for a thread $t_2$, the scheduler must ensure that $t_2$ is allowed to execute, thus freeing $t_1$ from waiting. If the system, for whatever reason, cannot guarantee that $t_2$ will eventually execute, then $t_1$ might wait indefinitely. A scheduler which guarantees the relative execution of threads is said have the forward progress property.

Currently, GPU specifications do not provide forward progress properties between threads in different work-groups, effectively disallowing popular existing idioms (and thus, programs) from multicore CPU systems to be ported to GPUs in a reliable manner. However, GPU programmers have discovered that by exploiting quirks in today’s GPUs, certain blocking idioms can be made to execute as expected. This can be achieved pragmatically by determining (via trial and error) the number of work-groups for which forward progress appears to be guaranteed, and hard coding this number into the GPU kernel. This can be fiddly, since in practice the number is influenced by the program (e.g. due to register usage), the target GPU (e.g. due to hardware resource limits), and the GPU driver (e.g. due to its policy on how resources are allocated). Moreover, this approach to writing applications suffers from two drawbacks. First, applications that exploit such quirks are not portable even across GPUs from the same designer: undesirable behaviour (e.g. indefinite spinning) may occur when executing the program on a different GPU model, applying otherwise semantics-preserving changes to the program, or updating the GPU driver. Second, because the behaviour is not guaranteed by GPU programming language specifications, and applications written in this manner are generally not tested on a GPUs from a wide range of vendors [41], an application tested on multiple GPU models from one vendor may nevertheless fail when executed on a GPU from another vendor. Indeed, we observed that an inter-work-group synchronisation barrier written for one GPU (e.g. the NVIDIA GTX Titan) might deadlock when run on a different GPU (e.g. the Intel HD5500).
To address the above drawbacks, we performed a large empirical study (across eight GPUs spanning four vendors) to search for a forward progress abstraction that was (a) able to account soundly for the behaviour across the GPUs that we tested, and (b) useful enough to allow the implementation of popular blocking idioms [42]. After probing our zoo of GPUs with litmus tests, we determined that all GPUs had the following property: once a work-group begins executing a kernel (i.e. the work-group becomes *occupant* on a hardware resource), it will continue executing until it reaches the end of the kernel. We call this execution model the *occupancy bound* execution model, because the number of work-groups for which relative forward progress is guaranteed is bound by the hardware resources available for executing work-groups; i.e. the hardware resources determine how many work-groups can be occupant.

The occupancy bound execution model naturally allows mutex synchronisation, as a thread that acquires a lock is guaranteed to continue executing, thus eventually releasing the lock. A portable synchronisation barrier across work-groups is, however, much more difficult to achieve. To address this, we developed a *discovery protocol*, which dynamically, at kernel launch, determines a safe, lower-bound estimate of the number of occupant work-groups. Work-groups that are not part of the initial wave of occupant work-groups discover that this is the case, and immediately exit the kernel. As a result, they do not participate in any blocking synchronisation. Kernels must be designed to be agnostic to the number of executing work-groups, to account for the fact that the number of occupant work-groups may vary across GPUs and between driver versions, and due to the fact that the discovery protocol does not guarantee discovering all work-groups that are initially occupant, and may discover a different number of work-groups during different executions of the same kernel. Previous work shows that it is usually possible to design kernels to satisfy this constraint, and programs with this property are said to use the *persistent thread* programming model [21].

The discovery protocol uses a polling mechanism, protected by a mutex, to allow work-groups to mark themselves as executing. Because the poll is only open for a finite amount of time, it is possible that the occupant work-groups may not be able to mark themselves, and contribute to the computation, thus under utilising the GPU. We show through a wide range of experiments that on current GPUs, we are able to define a discovery protocol that has a recall of nearly 100%. Our main insight is that using a *fair* mutex, such as a ticket lock, to guard the polling data structure leads to better recall than is achievable using a more traditional compare-and-swap-based spin-lock.

Open problems

As part of our work, we benchmarked a set of applications that use our discovery protocol and inter-work-group synchronisation barrier against versions of the same applications that use a *multi-kernel* approach to achieve global synchronisation (by explicitly ending and re-launching the kernel). We observed a wide range of performance profiles across the eight GPUs that we tested. For example, on Intel GPUs, our inter-work-group barrier provides a median 28% performance improvement compared with the multi-kernel approach, while on ARM GPUs we observed a median 50% *slowdown* using the inter-work-group barrier. An interesting open problem is to investigate whether a performance model can be developed that can predict when it is beneficial to use an inter-work-group barrier. Recent work has started to develop such a model for NVIDIA GPUs, but is yet to be extended to GPUs of other vendors [37].

---

4 Of course, it is possible as usual for deadlock to occur due to mismanagement of multiple mutexes.
In our work so far we have only examined the relative forward progress properties associated with threads in different work-groups. Forward progress properties between threads at different levels of the GPU execution hierarchy may not be the same. As with inter-work-group forward progress, GPU specifications do not provide much guidance with respect to intra-work-group forward progress. In this context we think the following are interesting research topics: (a) an empirical investigation of the forward progress properties that GPUs appear to provide at each level of the execution hierarchy, and (b) the identification of applications that might benefit from blocking idioms at additional levels of this hierarchy.

Finally, because GPU specifications do not provide forward progress guarantees at present, we believe that working with standards committees in this area is vital. Our understanding from talking with various contacts in the GPU industry is that one reason vendors are reluctant to provide forward progress guarantees is so that they can reserve the right to dynamically change the computational resources assigned to a given GPU kernel, e.g. to make computational resources available to other tasks, or to reduce energy consumption. In our most recent work we have proposed an extension to the GPU programming model called cooperative kernels, specifically designed with blocking algorithms in mind, which we hope may allow the forward progress requirements of blocking algorithms to co-exist with the need for dynamic changes in the hardware resources that are assigned to kernels [43].

Acknowledgements. The line of work on which we report was contributed to by many co-authors: Jade Alglave, Ethel Bardsley, Mark Batty, Bradford Beckmann, Adam Betts, Nathan Chong, Peter Collingbourne, Pantazis Deligiannis, Hugues Evrard, Ganesh Gopalakrishnan, Daniel Liew, Daniel Poetzl, Shaz Qadeer, Zvonimir Rakamarić, and Paul Thomson.

References
Forward Progress on GPU Concurrency


A. F. Donaldson, J. Ketema, T. Sorensen, and J. Wickerson

33 NVIDIA. CUDA C programming guide, version 5.5, 2013.
Admissibility in Games with Imperfect Information∗

Romain Brenguier1, Arno Pauly2, Jean-François Raskin3, and Ocan Sankur4

1 Oxford University, UK
2 Université Libre de Bruxelles, U.L.B., Belgium
3 Université Libre de Bruxelles, U.L.B., Belgium
4 CNRS – IRISA Rennes, France

Abstract
In this invited paper, we study the concept of admissible strategies for two player win/lose infinite sequential games with imperfect information. We show that in stark contrast with the perfect information variant, admissible strategies are only guaranteed to exist when players have objectives that are closed sets. As a consequence, we also study decision problems related to the existence of admissible strategies for regular games as well as finite duration games.


Keywords and phrases Admissibility, non-zero sum games, reactive synthesis, imperfect information

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2017.2

Category Invited Talk

1 Introduction

Two-player zero-sum perfect information games played on finite (directed) graphs are the canonical model to formalize the reactive synthesis problem [24, 1]. Unfortunately, this mathematical model is often a too coarse abstraction of reality. First, realistic systems are usually made up of several components, each of them with its own objective. These objectives are not necessarily antagonistic. Hence, the setting of non-zero sum graph games needs to be investigated, see [9] and additional references therein. Second, in systems made of several components, each component has usually a partial view on the entire system. Hence it is natural to study games with imperfect information [25, 13]. In this paper, we investigate the notion of admissible strategies for infinite duration non-zero sum games played on graphs in which players have imperfect information.

The objective W of a player in such a game is a set of infinite paths, those that model behaviors of the system that are regarded as satisfactory by the player. During the course of the game, players move a token from vertices to adjacent vertices. Each player applies one strategy which is a function that maps histories of plays that ends in a vertex owned by the player to adjacent vertices. The strategy instructs the player where to move the token
according to the prefix of play constructed so far. For a player with objective $W$, a strategy $\sigma$ is said to be dominated by a strategy $\sigma'$ if $\sigma'$ does as well as $\sigma$ with respect to $W$ against all the strategies of the other players and strictly better for some of them. A strategy $\sigma$ is admissible for a player if it is not dominated by any other of his strategies. Clearly, playing a strategy which is not admissible is sub-optimal, and as a consequence a rational player should favor admissible strategies.

While admissibility is a classical concept for games in normal form, i.e. matrix games, it was first studied in this context of infinite duration games by Berwanger in [6]. In that paper, it was shown that admissibility, i.e. the avoidance of dominated strategies, is well-behaved in infinite duration $n$-player non-zero sum turn-based games with perfect information and Boolean outcomes (two possible payoffs: win or lose). This framework encompasses games with omega-regular objectives. The main contributions of Berwanger were to show that:

(i) In all $n$-player game structures, for all objectives, players have admissible strategies. Berwanger even shows the existence of strategies that survive the iterated elimination of dominated strategies.
(ii) Every strategy that is dominated by a strategy is dominated by an admissible strategy.
(iii) For finite game structures, the set of admissible strategies forms a regular set.

Contributions

In this paper, we study the notion of admissible strategies in the more general setting of infinite duration games with imperfect information. We obtain results that are in stark contrast with the results obtained by Berwanger:

(i) In a 2-player game with imperfect information, players may have no admissible strategies at all, even for reachability objectives (Example 9). In particular, there are strategies that are dominated but not dominated by any admissible strategy.
(ii) Admissible strategies always exist for the class of closed objectives, i.e. safety winning conditions, and their existence is only guaranteed for this class of objectives (Theorem 10).
(iii) The set of admissible strategies of a player depends on the informedness of the other players: this set is the largest when the other players are perfectly informed (Theorem 17).
(iv) For the special case of finite duration games, we know by point (ii) that admissible strategies always exist, but we show that simple queries about this set are $NP$-complete (Theorem 28).
(v) For infinite duration 2-player games, we characterize precisely the set of admissible strategies when the second player is fully informed and we show how to decide the existence of admissible strategies (Theorem 20). When the second player is not fully informed but more informed than the first player, then we show how to decide the existence of admissible strategies for the first player by a reduction to the model-checking problem of Strategy Logic with (hierarchical) imperfect information (Theorem 32). In the general case, we show how to decide, given a regular observation-based strategy if this strategy is admissible or not (Theorem 35). For this last case, we left open the decidability status of the non-emptiness problem for the set of admissible strategies.
Related works

The iterated elimination of dominated strategies formalizes a strong notion of rationality [2]. In [18], Faella considers games played on finite graphs and focuses on the vertices from which one designated player cannot force a win. He compares several criteria for establishing what is the preferable behavior of this player from those vertices, eventually settling on the notion of admissible strategy. In [11], starting from the notion of admissible strategy, we have defined a novel rule for the compositional synthesis of reactive systems, applicable to systems made of $n$ components which have each their own objective. We have shown that this synthesis rule leads to solutions which are robust and resilient. In [12], we have shown how to solve the central algorithmic questions related to admissibility for omega-regular objectives. In all those works, players are assumed to have perfect information.

In [10], we have studied the notion of admissible strategies for non-zero quantitative games with perfect information. Similarly to games with imperfect information, in the quantitative case, admissible strategies may not exist. Nevertheless, in all games where the adversarial and the cooperative values can be realized, i.e. games in which there exist worst-case optimal strategies and cooperative optimal profiles, players are guaranteed to have admissible strategies.

In [4], we have studied the notion of admissibility for concurrent games. Concurrent games [17] are $n$-player games played on graphs in which players take moves concurrently: at each round, all players choose an action at the same time and without informing the other players. This set of actions determines the next vertex. Because players choose actions without being informed of the concurrent choices of the other players, concurrent games are a special case of imperfect information games. In concurrent games, in contrast with general games with imperfect information studied here, admissible strategies are guaranteed to exist. While in this paper, we limit our study to deterministic strategies, in [4], we have also considered the more general case of randomized strategies and the notion of almost-sure winning.

In [16], Damm and Finkbeiner use the notion of dominant strategy to provide a compositional semi-algorithm for the (undecidable) distributed synthesis problem. So while we use the notion of admissible strategy, they use a notion of dominant strategy. The notion of dominant strategy is strictly stronger: every dominant strategy is admissible but an admissible strategy is not necessary dominant.

In normal form games it is trivial to decide whether a strategy dominates another one, or is dominated by an arbitrary strategy. Iterated elimination of dominated strategies does yield a relevant computational problem here, though. Depending on the type of dominance used (weak dominance, dominance, strict dominance) and the number of distinct payoffs, these problems were shown to be complete for L, NL, P or NP in [23, 15, 20, 8]. See [23] for an overview.

Structure of the paper

In Section 3, we discuss the existence of admissible strategies in two player win/lose infinite sequential games with imperfect information. In Section 4, we discuss the impact of informedness of the opponent on the set of admissible strategies. In Section 5, we characterize the set of admissible strategies when the opponent is perfectly informed. In Section 6, we consider the special case of games on finite trees. In Section 7, we investigate decision problems related to dominance in regular games of infinite duration.
2 Definitions

Games of imperfect information

Given some finite word $w$, let $\ell(w)$ denote its last element and $|w|$ its length. For an finite or infinite word $w$, if $|w| \geq n$, let $w_{\leq n}$ denote the prefix of $w$ of length $n$, and for $i, 0 \leq i < |w|$, let $w(i)$ denote the letter in position $i$ in $w$. Given an infinite word $w$, let $\inf(w)$ denote the letters in $w$ that appear infinitely often along $w$.

Definition 1. A two player win/lose infinite sequential game with imperfect information (short: infinite game with imperfect information) between Player 1 (sometimes called the protagonist) and Player 2 (sometimes called the opponent) $G = (d, W, \Xi_1, \Xi_2)$ is given by the following:
1. a winning condition $\{0, 1\}^* \rightarrow \{1, 2\}$ assigning a player to each vertex in the full infinite binary tree.
2. a control partition $W \subseteq \{0, 1\}^\omega$ for the first player.
3. equivalence relations $\Xi_i$ on $\{0, 1\}^*$ for $i \in \{1, 2\}$ satisfying the following properties:
   a. $v \Xi_i u \Rightarrow d(v) = d(u)$ (knowledge of who is playing is guaranteed).
   b. If $v \not\Xi_i u$, then $v b \Xi_i u b'$ for any $b, b' \in \{0, 1\}$ (perfect recall).
   c. $v \Xi_i u \Rightarrow |v| = |u|$ (ability to count moves).

Player $i$ is perfectly informed if $\Xi_i$ is the identity noted $id_i$. We only specify an objective for Player 1 as we will characterize the admissible strategies of this player, and to do this, the objective of Player 2 is irrelevant.

Definition 2. Elements of $\{0, 1\}^*$ (resp. $\{0, 1\}^\omega$) are called finite plays or histories (resp. plays). A strategy of Player $i$ is a function $\sigma : d^{-1}(i) \rightarrow \{0, 1\}$. It is observation-based, if $v \Xi_i u$ implies $\sigma(v) = \sigma(u)$. A play $p \in \{0, 1\}^\omega$ is compatible with a strategy $\sigma$ of Player $i$, if for all $n \in \mathbb{N}$, if $d(p_{\leq n}) = i$, then $\sigma(p_{\leq n}) = p(n + 1)$. We write $\Sigma_1 (\Sigma_{1}^\omega)$ and $\Sigma_2 (\Sigma_{2}^\omega)$ for the set of all (observation-based) strategies for Player 1 and Player 2 respectively.

Definition 3. Given strategies $\sigma \in \Sigma_1$, $\tau \in \Sigma_2$, we let $\Out(\sigma, \tau)$ be the unique $p \in \{0, 1\}^\omega$ compatible with both $\sigma$ and $\tau$. A strategy $\sigma \in \Sigma_{1}^\omega$ of Player 1 is an adversarially winning strategy in $G$ if $\forall \tau \in \Sigma_2 : \Out(\sigma, \tau) \in W$, we denote this by $G, \sigma \models W$. A strategy $\sigma \in \Sigma_{1}^\omega$ is cooperatively winning if $\exists \tau \in \Sigma_{2}^\omega : \Out(\sigma, \tau) \in W$.

Definition 4. A regular game $R(v_0) = (V, E, V_1, V_2, O^1, O^2, \Omega)$ is given by
1. a finite directed graph $(V, E)$ with a designated starting vertex $v_0 \in V$ where all vertices have out-degree 1 or 2, moreover, the successors of each vertex $v \in V$ are ordered, and denoted by $s(v, 0)$ and $s(v, 1)$ (if $v$ has out-degree 1, they are equal).
2. a control partition $V = V_1 \cup V_2$, $V_i$ being the vertices controlled by Player $i$, $v_0$ is the starting vertex.
3. two observation partitions $V = \bigcup_{i \in \{1, 2\}} O_i$ for $i \in \{1, 2\}$ which are both refinements of the control partition. We denote by $O_i^{\text{turn}}$ the set of observations of Player $i$ that contain vertices that belong to Player $i$.
4. an $\omega$-regular winning condition $\Omega \subseteq V^\omega$ defined by a parity condition $pr : V \rightarrow \{1, 2, \ldots, d\}$ and such that $\rho \in \Omega$ if and only if $\min\{k \mid v \in \inf(\rho) \land \rho(v) = k\}$ is even.

Remark. For convenience, we will give examples of regular games violating the condition that the out-degree is 1 or 2. In this case, we understand implicitly that the following
modifications are employed to satisfy this condition: If \(2^d\) is an upper bound for the out-degree, then any non-sink vertex is replaced by a full binary tree of height \(d\), with the partitions and winning condition being extended accordingly.

Definition 5. From a regular game \(R(v_0)\), we obtain an infinite sequential game \(R^\omega(v_0) = (d, W, \forall_0, \exists_1)\) by unfolding as follows:

1. Let \(\epsilon : \{0, 1\}^* \to V^*\) be defined via \(\epsilon(\epsilon) = v_0\) and \(\epsilon(wb) = \epsilon(w) \cdot s(\ell(\epsilon(w)), b)\). Let \(d(w) = i\) if \(\ell(\epsilon(w)) \in V_i\).
2. Let \(\tilde{\epsilon} : \{0, 1\}^\omega \to V^\omega\) be the limit induced by \(\epsilon\). Let \(W = \tilde{\epsilon}^{-1}(\Omega)\).
3. Let \(\equiv_i\) be the smallest equivalence relation that satisfies the following constraints:
   a. If \(w \equiv_i w'\) and \(d(w) = 3 - i\) and \(\exists j \; \ell(i(wb)) \in O_j \land \ell(i(w'b)) \in O_j\), then \(wb \equiv_i w'b\).
   b. If \(w \equiv_i w'\) and \(d(w) = i\) and \(\exists j \; \ell(i(wb)) \in O_j \land \ell(i(w'b)) \in O_j\), then \(wb \equiv_i w'b\).

We also consider the special case in which Player 2 is perfectly informed. In that case, we do not take into account the partition \(O^2\) and \(R^\omega(v_0) = (d, W, \forall_1, \exists_2)\) is defined as above but \(\equiv_2 = id_2\) is the identity relation.

To ease presentation, we make a limited use of computation tree logic (CTL) to state simple facts on strategies in regular games. We refer the interested reader for instance to [3] for formal definitions. Recall that for subsets \(A, B \subseteq V, FA\) (resp. \(GA\)) is the set of plays that visit \(A\) (resp. that always stay in \(A\)). We use the following CTL formulas. \(G, \sigma \models FA\) means that all plays compatible with \(\sigma\) eventually reach \(A\). Furthermore \(G, \sigma \models EF\sigma\) means that there exists \(\tau \in \Sigma_0^{\omega}\) such that \(\nu(Out(\sigma, \tau))\) eventually reaches \(A\).

For a subset \(A \subseteq V\) of vertices, \(GA\) denotes the game that starts with nondeterministically moving to any vertex of \(A\), while both players receive respective observations of the chosen vertex. Strategy \(\sigma\) is winning from \(v \in V\) (resp. \(A \subseteq V\)) if it is winning in \(G([v])\) (resp. \(G(A)\)).

Definition 6. We consider a regular game \(R(v_0)\). Given some finite sequence or infinite sequence \(w \in \{0, 1\}^{\leq \omega}\), the derived observation sequence for Player \(i\) is \(O_i(w) = O_i^0 \cup O_i^1 \cup \ldots\) where \(\nu_i(w(k)) \in O_i^k\) for all \(k \leq |w|\). An observation-based strategy for Player \(i\) in \(R(v_0)\) is equivalent to a function \(\sigma : (O_i)^* \cdot O^\omega_{turn} \to \{0, 1\}\). Now a strategy \(\sigma\) of Player \(i\) is regular, if there exists a finite deterministic automaton \(A = (Q^A, q_0^A, O^i, \delta^A, F^A)\), where \(Q^A\) is a finite set of states, \(q_0^A\) is the initial state, \(O^i\) is the alphabet, \(\delta : Q^A \times O^i \to Q^A\), and \(F^A \subseteq Q^A\) is the set of accepting states, and the observation-based strategy \(\sigma\) is encoded by the language of the automaton as follows: when reading words \(O_{turn}^i \cup O_{turn}^j \cup \ldots\), and whenever \(d(w) = i\), then the automaton accepts the observation sequence \(O_i(w)\) induced by \(w\) iff \(\sigma(O_i(w)) = 0\).

Definition 7. Given a regular game \(R(v_0)\), the knowledge of Player \(i\) playing the observation-based strategy \(\sigma\) after a sequence of observations \(\rho = o_1o_2\ldots o_n \in (O_i)^*\), denoted \(K(\sigma, \rho)\), is the set of vertices \(v\) such that there exists \(w \in \{0, 1\}^*\):

1. \(O^i(\nu_i(w)) = \rho\)
2. for all \(j, 1 \leq j < n\), if \(o_j \in O^i_{turn}\), then \(\sigma(\rho_{\leq j}) = w_j\)
3. \(v\) is equal to \(\nu_i(w)\).

Dominance and admissibility

Definition 8. For \(\sigma, \sigma' \in \Sigma_0^{\omega}\), we say that \(\sigma'\) weakly dominates \(\sigma\) and write \(\sigma' \succeq \sigma\) if \(\forall \tau \in \Sigma_0^{\omega}, \text{Out}(\sigma, \tau) \in W \Rightarrow \text{Out}(\sigma', \tau) \in W\). If in addition \(\exists \tau \in \Sigma_0^{\omega}, \text{Out}(\sigma, \tau) \notin W\), and \(\text{Out}(\sigma', \tau) \in W\), then \(\sigma'\) dominates \(\sigma\) and write \(\sigma' \succeq \sigma\). We say that a strategy \(\sigma \in \Sigma_0^{\omega}\) is admissible if there is no \(\sigma' \in \Sigma_0^{\omega}\) such that \(\sigma' \succeq \sigma\). Note that if \(\sigma \preceq \sigma'\) but \(\sigma \neq \sigma'\), then \(\sigma' \preceq \sigma\), justifying the notation.
Admissibility in Games with Imperfect Information

Let the existence of admissible strategies, provided we use colours to define the winning condition. Hence there is no observation-based admissible strategy for Player 1 in this game, and

\[ \sigma \]

Example 9. We exhibit a regular game with a reachability objective (to reach the state marked 'w') lacking observation-based admissible strategies for Player 1. The graph is depicted in Figure 1. Player 1 controls circle vertices, Player 2 controls box vertices. Player 1 wins all plays that eventually enter the vertex 'w'. Player 1 has imperfect information: he can not differentiate between 's1' and 's2', and not between 't1' and 't2'. As a consequence, the observation-based strategies available to Player 1 are essentially the strategies \( \sigma_n \), one for each \( n \in \mathbb{N} \), which play the action '0' for \( n \) consecutive steps followed by the action '1', and \( \sigma_\infty \), which always plays the action '0'. Then it is easy to show that \( \sigma_\infty \prec \sigma_0 \prec \sigma_1 \ldots \), hence there is no observation-based admissible strategy for Player 1 in this game.

![Figure 1](image_url) Player 1 can not differentiate between 's1' and 's2', and not between 't1' and 't2'. In this example, no strategy is admissible for Player 1.

3 Existence of Admissible Strategies

A starting observation for our investigation is that the existence of admissible strategies in infinite sequential games with imperfect information is not guaranteed, even for regular games and reachability objectives. This is in stark contrast to the case where both players are perfectly informed [6].

Example 9. We exhibit a regular game with a reachability objective (to reach the state marked 'w') lacking observation-based admissible strategies for Player 1. The graph is depicted in Figure 1. Player 1 controls circle vertices, Player 2 controls box vertices. Player 1 wins all plays that eventually enter the vertex 'w'. Player 1 has imperfect information: he can not differentiate between 's1' and 's2', and not between 't1' and 't2'. As a consequence, the observation-based strategies available to Player 1 are essentially the strategies \( \sigma_n \), one for each \( n \in \mathbb{N} \), which play the action '0' for \( n \) consecutive steps followed by the action '1', and \( \sigma_\infty \), which always plays the action '0'. Then it is easy to show that \( \sigma_\infty \prec \sigma_0 \prec \sigma_1 \ldots \), hence there is no observation-based admissible strategy for Player 1 in this game.

We can characterize exactly the properties of the winning condition \( W \) that ensures the existence of admissible strategies, provided we use colours to define the winning condition. Let \( C \) be a finite set of colours, and \( c : \{0, 1\}^\omega \rightarrow C \) be a colouring function. Let \( \hat{c} : \{0, 1\}^\omega \rightarrow C^\omega \) be defined via \( \hat{c}(p)(n) = c(p_{\leq n}) \). We say that \( W \subseteq \{0, 1\}^\omega \) is induced by \( W_C \subseteq C^\omega \) if there exists a colouring function \( c \) such that \( W = \hat{c}^{-1}(W_C) \). Note that functions of the form \( \hat{c} \) are exactly the 1-Lipschitz functions from \( \{0, 1\}^\omega \) to \( C^\omega \).

Theorem 10. The following are equivalent for \( W_C \subseteq C^\omega \):
1. \( W_C \) is closed (i.e. corresponds to a safety condition).
2. All imperfect information games with winning set induced by \( W_C \) have admissible strategies.

Our proof of this theorem heavily relies on topological arguments. Note that we can conceive of \( \Sigma_1^{\geq i} \) as a metric space by setting \( d(\sigma, \sigma') = 2^{-k} \) for the least \( k \in \mathbb{N} \) such that \( \exists w \in \{0, 1\}^k \cap \sigma^{-1}(i) \) with \( \sigma(w) \neq \sigma'(w) \). With this metric, \( \Sigma_1^{\geq i} \) is a compact metric space, hence every sequence of strategies has an accumulation point. Moreover, the map \( (\sigma, \tau) \mapsto \text{Out}(\sigma, \tau) : \Sigma_1^{\geq i} \times \Sigma_2^{\geq i} \rightarrow \{0, 1\}^\omega \) is continuous. We will write \( B(\sigma, k) := \{ \sigma' \in \Sigma_1^{\geq i} \mid d(\sigma, \sigma') < 2^{-k} \} \) and utilize the fact that there are only countably many distinct sets \( B(\sigma, k) \).

Lemma 11. Let \( W \) be closed. Let \( (\sigma_n)_{n \in \mathbb{N}} \) be a sequence in \( \Sigma_1^{\geq i} \) with \( \sigma_n \preceq \sigma_{n+1} \) for all \( n \), and let \( \sigma \) be an accumulation point of \( (\sigma_n)_{n \in \mathbb{N}} \). Then \( \forall n \in \mathbb{N} : \sigma_n \preceq \sigma \).
Assume that for some \( R. \) Brenguier, A. Pauly, J.-F. Raskin, and O. Sankur 2:7

Proof. Assume that for some \( i \in \mathbb{N} \) we have \( \sigma_i \not\sim \sigma \). Then there is some \( \tau \in \Sigma_2^\mathbb{N} \) such that \( \text{Out}(\sigma, \tau) \not\in W \) and \( \text{Out}(\sigma_n, \tau) \in W \). As \( W \) is closed, \( \text{Out}(\sigma, \tau) \not\in W \) depends only on some finite prefix of \( \sigma \), i.e. whenever \( d(\sigma, \sigma') < 2^{-k} \) for a suitably large \( k \), then also \( \text{Out}(\sigma', \tau) \not\in W \).

As \( \sigma \) is an accumulation point of \((\sigma_n)_{n \in \mathbb{N}} \), there is some \( j \geq i \) with \( d(\sigma, \sigma_j) < 2^{-k} \).

By assumption and transitivity of \( \preceq \), we find that \( \sigma_i \preceq \sigma_j \). But \( \text{Out}(\sigma_i, \tau) \in W \), whereas \( \text{Out}(\sigma_j, \tau) \not\in W \) – contradiction.

\[ \text{Corollary 12.} \] Let \((\sigma_n)_{n \in \mathbb{N}}\) be a sequence of protagonist strategies with \( \sigma_n \prec \sigma_{n+1} \) for all \( n \), and let \( \sigma \) be an accumulation point of \((\sigma_n)_{n \in \mathbb{N}} \). Then \( \forall n \in \mathbb{N} \, \sigma_n \prec \sigma \).

\[ \text{Lemma 13.} \] Let \((\sigma_\beta)_{\beta < \alpha} \) be an ordinal-indexed sequence of protagonist strategies such that \( \sigma_\beta \prec \sigma_\gamma \) for any \( \beta < \gamma < \alpha \). Then \( \alpha \) is countable.

Proof. By assumption, \( p_\gamma \not\prec p_\beta \) for any \( \gamma > \beta \). This is witnessed by some \( \tau \in \Sigma_2^\mathbb{N} \) such that \( \text{Out}(\sigma_\gamma, \tau) \in W \) and \( \text{Out}(\sigma_\beta, \tau) \not\in W \). As \( W \) is closed, there is some \( k \in \mathbb{N} \) such that \( \text{Out}(\sigma_\beta, \tau') \not\in W \) for all \( \tau' \in B_{\beta, \gamma} := B(\tau, k) \).

As there are only countably many distinct sets of the form \( B(\tau, k) \), if \( \alpha \) were uncountable, there would have to be \( \beta < \gamma < \beta' < \gamma' \) with \( B_{\beta, \gamma} = B_{\beta', \gamma'} \). By construction we have that \( \exists \tau \in B_{\beta, \gamma} : \text{Out}(\sigma_\gamma, \tau) \in W \) and \( \forall \tau \in B_{\beta', \gamma'} : \text{Out}(\sigma_\beta, \tau) \not\in W \). But this contradicts that \( \sigma_\gamma \preceq \sigma_\beta \), hence \( \alpha \) has to be countable.

\[ \text{Lemma 14.} \] If \( W_C \subseteq C^\omega \) is not closed, then there is an imperfect information game with winning set induced by \( W_C \) without an admissible strategy.

Proof. This is a generalization of the Example 9. As \( W_C \) is not closed, we can pick a sequence \((p_n)_{n \in \mathbb{N}}\) and path \( p_\infty \) in \( C^\omega \) such that \( d(p_n, p_\infty) < 2^{-2n} \), \( p_n \in W_C \) and \( p_\infty \not\in W_C \).

The two players take turns, i.e. \( d(w) = 1 \) iff \( |w| \) is even. It is unknown to Player 1 which moves Player 2 has taken, i.e. \( \cong_1 \) is the coarsest equivalence relation satisfying the criteria. The colouring ensures that any sequence of the form \( 0^{j+1}1w1q \) is mapped to \( p_{j+l+1} \) where \( |w| = 2l \) and \( w \) contains no \( 1 \) in an odd position, while any other sequence gets mapped to \( p_\infty \). The sequence \((p_n)_{n \in \mathbb{N}}\) and \( p_\infty \) were chosen in a way to make this possible.

Intuitively, this means that Player 1 wins iff he plays a \( 1 \) for the first time at some turn after Player 2 already has played a \( 1 \). As Player 1 cannot observe the moves of Player 2, we find that as in Example 9 the observation based strategies available to Player 1 are essentially \( \sigma_n \), which plays \( 0 \) the first \( n \) times it is Player 1’s turn and then \( 1 \), and \( \sigma_\infty \), which always plays \( 1 \). Then again \( \sigma_\infty \prec \sigma_0 \prec \sigma_1 \ldots \), hence there is no admissible strategy.

Proof of Theorem 10. If \( W_C \) is closed, then any \( W \) induced by it is closed, too. We can start with some strategy \( p_0 \), and if it is dominated, move to some strategy \( p_1 \) it is dominated by, etc., and create a strictly increasing sequence \((p_n)_{n \in \mathbb{N}}\), unless we hit a non-dominated strategy. Then pick some accumulation point \( p_\omega \) (as the space of strategies is compact).

If \( p_\omega \) is dominated, pick a witness dominating strategy \( p_{\omega+1} \), etc. If we would never reach a non-dominated strategy, then this would by Lemma 11 create an increasing \( \Omega_1 \)-sequence (where \( \Omega_1 \) is the first uncountable ordinal), but this would contradict Lemma 13.

The second implication is the statement of Lemma 14.

Some further remarks

\[ \text{Observation 15.} \] In infinite trees, there are strictly increasing sequences of any countable length.
Proof. By induction. If $T$ is a tree with a strictly increasing sequence $(p_{\beta})_{\beta<\alpha}$ of length $\alpha$, let opponent choose between playing in $T$ or letting protagonist choose between losing and winning. Starting with protagonist choosing to lose outside of $T$, and playing $p_0$ inside $T$, he can improve inside $T$ for $\alpha$ steps and then decide to win if outside of $T$, yielding an improvement sequence of length $\alpha + 1$.

If for each $i \in \mathbb{N}, T_i$ is a tree with an improvement sequence $(p_{\beta})_{\beta<\alpha_i}$, then by letting opponent choose in which tree to play (or non at all), we can obtain an improvement sequence of length $\sup_{i \in \mathbb{N}} \alpha_i$.

Observation 16. The construction in Lemma 14 is sufficiently uniform that the preceding argument also shows that for some $\omega$-regular winning condition $W$ all imperfect information games played on finite graphs have admissible strategies iff $W$ is a safety condition.

4 The Impact of the Informedness of the Opponent

The following theorem characterizes how the information available to the opponent (Player 2) impacts dominance between strategies of Player 1. When the second player is less informed then there are more dominated strategies for Player 1 and so less admissible strategies for Player 1.

We consider two infinite sequential games with imperfect information that only differ in the equivalence relation of Player 2, with one having $\equiv_2$ and the other having $\equiv'_2$. The set of (observation-based) strategies for Player 1 in both games is the same. We assume that $\equiv_2$ is coarser than $\equiv'_2$, i.e. $w \equiv'_2 w' \Rightarrow w \equiv_2 w'$. Then $\Sigma^\equiv'_2 \supseteq \Sigma^\equiv_2$. We write $\sigma \prec \sigma'$ ($\sigma \prec' \sigma'$) if $\sigma$ is dominated by $\sigma'$ in the game built with $\equiv_2$ (with $\equiv'_2$).

Theorem 17. $G = (d, W, \equiv_1, \equiv_2)$ and $G' = (d, W, \equiv_1, \equiv'_2)$, where $\equiv_2$ is coarser than $\equiv'_2$, let $\prec$ be the dominance ordering in $G$ and $\prec'$ in $G'$, for all observation-based strategies $\sigma$ and $\sigma'$ of Player 1 if $\sigma \prec' \sigma'$, then $\sigma \prec \sigma'$.

Proof. By the definition of $\leq$ and $\leq'$, we see that $\Sigma^\equiv'_2 \supseteq \Sigma^\equiv_2$ implies that if $\sigma \leq' \sigma'$, then $\sigma \leq \sigma'$. To extend this to $\prec$ and $\prec'$, we need show that if there is some strategy $\tau' \in \Sigma^\equiv'_2$ such that $\text{Out}(\sigma, \tau') \notin W$ but $\text{Out}(\sigma', \tau') \in W$, then there already is some $\tau \in \Sigma^\equiv_2$ with this property.

Consider only word-strategies played by Player 2, i.e. strategies not depending on the actions of the opponent at all. If there is a word-strategy obtaining different outcomes against $\sigma$ and $\sigma'$, which by assumption of $\sigma \leq \sigma'$ can only mean that the outcome of the word strategy against $\sigma$ is not in $W$ while the outcome of the word strategy against $\sigma'$ is in $W$, we would be done – as all word-strategies are in $\Sigma^\equiv_2$, the requirement that players can count the number of moves played so far.

If $\tau$ is played against $\sigma$, it will act like some particular word strategy (denoted by $\tau^\sigma$), likewise $\tau$ acts like the word strategy $\tau^\sigma$ when played against $\sigma'$. If $\tau^\sigma = \tau^\sigma'$, then this is the witness we are looking for. Otherwise, $\tau^\sigma = h \rho$ and $\tau^\sigma' = h' \rho'$ for some maximal common prefix $h$, and after playing $h$ against either $\sigma$ or $\sigma'$, Player 2 can distinguish w.r.t. $\equiv'_2$ which of the two strategies he is playing against. Thus, the following strategy $\tau'$ is also in $\Sigma^\equiv_2$: Play $h$. If faced against $\sigma$, play $\rho$, if faced against $\sigma'$, play $\rho'$. Now $\tau'$ behaves like $\tau^\sigma$ if played against $\sigma$, and like $\tau^\sigma'$ if played against $\sigma'$. If all word strategies would yield the same outcomes against $\sigma$ and $\sigma'$, then $\tau'$ wins against $\sigma'$ and loses against $\sigma$, a contradiction to $\sigma \prec' \sigma'$. Thus, there has to be a word strategy yielding different outcomes, which as explained above, provides the witness for $\sigma \prec \sigma'$. △
Corollary 18. If a strategy is not admissible w.r.t. perfect information strategies of the opponent, then it is not admissible w.r.t. observation-based strategies of the opponent.

To conclude this section, we shall demonstrate that Player 2 having imperfect information can indeed lead to strictly less admissible strategies for Player 1 compared to a perfectly informed Player 2:

Example 19. Consider a game where Player 1 moves first and plays a or b. Then Player 2 responds with x, y or z. Player 1 wins with the combinations ax, bx and by. If Player 2 can observe the move of Player 1, then both strategies of Player 1 are admissible. If Player 2 cannot observe the first move, then only playing b is admissible for Player 1.

5 Characterizing Admissibility with Perfectly Informed Opponents

We will provide a characterization of the admissible strategies of Player 1 under the assumption that Player 2 is perfectly informed. Corollary 18 shows that this is in a sense a conservative information, as any strategy found to be not admissible here will not be admissible against arbitrary opponents.

Fix some \( \sigma \in \Sigma^*_1 \). We say that \( M \subseteq \{0,1\}^* \) is a \( \sigma \)-monochromatic set, if \( M \) is maximal under the two following constraints:

1. For any \( v, u \in M \) we find that \( v \equiv_1 u \).
2. Any \( v \in M \) is compatible with \( \sigma \).

Given some \( \sigma \)-monochromatic set \( M \), we partition \( M \) into \( M_w^\sigma \cup M_c^\sigma \cup M_f^\sigma \) such that:

- \( M_w^\sigma = \{ v \in M \mid \forall \tau \in \Sigma_2 : v \text{ is compatible with } \tau \Rightarrow \text{Out}(\sigma, \tau) \in W \} \), i.e. the set of histories in \( M \) from which the strategy \( \sigma \) is adversarially winning;
- \( M_c^\sigma = \{ v \in M \mid \exists \sigma, \tau : v \text{ is compatible with } \tau \land \text{Out}(\sigma, \tau) \notin W \} \), i.e. the set of histories in \( M \) from which the strategy \( \sigma \) is cooperatively winning;
- \( M_f^\sigma = \{ v \in M \mid \forall \tau \in \Sigma_2 : v \text{ is compatible with } \tau \Rightarrow \text{Out}(\sigma, \tau) \notin W \} \), i.e. the set of histories in \( M \) from which the strategy \( \sigma \) is losing against all the strategies of Player 2.

Note that in the definition of monochromatic set, we quantify over the entire set of strategies of Player 2 as Player 2 is assumed to be perfectly informed.

Theorem 20. An observation-based strategy \( \sigma \in \Sigma^*_1 \) of Player 1 is dominated, when Player 2 is perfectly informed, if and only if there exists a \( \sigma \)-monochromatic set \( K \) such that

1. there exists another strategy \( \sigma' \in \Sigma^*_1 \) such that \( K \) is also \( \sigma' \)-monochromatic,
2. the strategies \( \sigma \) and \( \sigma' \) induce partitions \( (K_w^\sigma, K_c^\sigma, K_f^\sigma) \) and \( (K_w'^\sigma, K_c'^\sigma, K_f'^\sigma) \) of \( K \) such that:
   
   a) \( K_w^\sigma \cup K_c^\sigma \subseteq K_w'^\sigma \);
   b) and either
      
      i) \( K_f^\sigma \cap (K_w'^\sigma \cup K_c'^\sigma) = \emptyset \),
      ii) or \( K_f^\sigma \neq \emptyset \).

Proof. We first establish the right to left direction. We construct an observation-based strategy \( \sigma'' \) that dominates \( \sigma \) when Player 2 is perfectly informed. The strategy \( \sigma'' \) is defined as follows. In all histories that are not extensions of those in \( K \), \( \sigma'' \) plays exactly as \( \sigma \). For all histories extending some history in \( K \), then \( \sigma'' \) plays as \( \sigma' \). Let us prove that \( \sigma'' \) has all the required properties to dominate \( \sigma \):

- First, we note that \( \sigma'' \) is observation-based as \( \sigma \) and \( \sigma' \) are both observation-based, and extensions of \( K \) cannot be \( \equiv_1 \)-equivalent to non-extensions of \( K \).
Second, let \( \pi \) be a strategy of Player 2, and assume that \( \text{Out}(\sigma, \pi) \in W \). Then we need to show that \( \text{Out}(\sigma', \pi) \in W \). We consider two different cases:

1. If no prefix of \( \text{Out}(\sigma, \pi) \) is in \( K \) then we have that \( \text{Out}(\sigma, \pi) = \text{Out}(\sigma', \pi) \) so \( \text{Out}(\sigma', \pi) \in W \).
2. If a prefix \( w \) of \( \text{Out}(\sigma, \pi) \) is in \( K \), then once we reach this prefix \( \sigma'' \) is now acting as \( \sigma' \).

As \( \text{Out}(\sigma, \pi) \in W \), \( w \) is such that \( w \in K_{w}^{\sigma} \cup K_{w}^{e} \), then we can conclude that \( w \in K_{w}^{\sigma'} \), and so \( \text{Out}(\sigma', \pi) \in W \).

Third, we need to establish the existence of one strategy \( \pi \) of Player 2 such that \( \text{Out}(\sigma', \pi) \in W \) and \( \text{Out}(\sigma, \pi) \not\in W \). For that we consider a history \( w \in K \) such that either:
   (i) \( w \in K_{w}^{e} \) (and so \( w \not\in K_{w}^{e} \)), or
   (ii) \( w \in K_{w}^{f} \land w \in K_{w}^{\sigma''} \cup K_{w}^{\sigma'''} \). Such a \( w \) always exists by hypothesis. We design \( \pi \) as follows. First, \( \pi \) plays compatible with \( w \). As \( \sigma \) and \( \sigma'' \) disagree at the history \( w \), w.l.o.g. let us consider that \( \sigma \) plays 0 while \( \sigma' \) plays 1. As Player 2 is perfectly informed, he observes this and behaves in a way that:
   (i) if \( w \in K_{w}^{e} \) and thus \( w \in K_{w}^{e'} \), then \( \pi \) does not help \( \sigma \) (this is possible as \( w \not\in K_{w}^{e} \)), and ensures that the outcome is outside \( W \), while \( \sigma'' \) behaves like \( \sigma' \) and is thus winning from \( w \) no matter what \( \pi \) is.
   (ii) if \( w \in K_{w}^{f} \land w \in K_{w}^{\sigma''} \cup K_{w}^{\sigma'''} \), then \( \pi \) helps Player 1 when he plays \( \sigma'' \) to ensure that the outcome is in \( W \) and as \( w \in K_{w}^{e} \), we know that the outcome when \( \sigma \) is played is not in \( W \).

We now establish the left to right direction. Assume that \( \sigma \) is dominated by some other strategy \( \sigma'' \). By definition of dominance, there exists a strategy \( \pi \) of Player 2 such that \( \text{Out}(\sigma, \pi) \not\in W \) and \( \text{Out}(\sigma'', \pi) \in W \). Let \( w \) be the longest common prefix of \( \text{Out}(\sigma, \pi) \) and \( \text{Out}(\sigma'', \pi) \), and let \( K \) be the \( \sigma \)-monochromatic set containing \( w \) (\( K \) is then also \( \sigma'' \)-monochromatic). Let us consider the partition of \( K \) for \( \sigma \) and \( \sigma'' \). First, we know that \( w \not\in K_{w}^{e} \) and \( w \in K_{w}^{\sigma''} \cup K_{w}^{\sigma'''} \). Also, because \( \sigma \) is dominated by \( \sigma'' \) then we have that:
   (i) for all \( v' \in K_{w}^{e} \), \( v' \in K_{w}^{\sigma''} \),
   (ii) for all \( v' \in K_{w}^{e} \), \( v' \in K_{w}^{\sigma''} \),
   (iii) for all \( v' \in K_{w}^{f} \), \( v' \in K_{w}^{\sigma''} \),

Otherwise, it is easy to obtain a contradiction with the fact that \( \sigma \) is dominated by \( \sigma'' \). All this implies the right properties on the respective partitions.

Remark. This characterization of dominance is particularly useful in the case of regular games. There the question whether for some monochromatic \( K \) there exists a strategy \( \sigma \) inducing a particular partition \( (K_{w}^{e}, K_{w}^{f}, K_{w}^{e'}) \) depends only on the set of last vertices of the histories of \( K \). This is because winning sets in regular games are defined with parity conditions that lead to prefix independence. In particular, there are only finitely many cases to check. We will exploit this algorithmically in Subsection 7.2.

6 Games on Finite Trees

The simplest non-trivial subclass of infinite sequential games with imperfect information are those with clopen winning sets \( W \). Essentially, this means that whether or not \( p \in W \) for some run \( p \) only depends on some fixed-length prefix of \( p \). Hence, we can consider these as finite tree games. Alternatively, we could conceive of these as regular games where the underlying graph is a tree.

The initial vertex is the root of the tree, and we denote by \( \text{Leaves}(G) \) the leaves of the tree, i.e. the finite histories determining membership in \( W \) of any run passing through them. W.l.o.g., we assume that all players know when a leaf is reached, and that both players can distinguish winning from losing leaves; in other terms, no \( \Xi_{e} \)-equivalence class contains both
a leaf and a non-leaf, and no class contains both a winning and a losing leaf. We denote the set of winning leaves by \( \Omega_f \).

In this section, for convenience, we assume that each vertex of Player \( i \) has \( k_i \) successors for a given \( k_i \geq 1 \), and that these successors are labeled with numbers between 0 and \( k_i - 1 \). We sometimes call these numbers actions, and denote Act\(_i\) = \{0, 1, \ldots, k_i\}, and Act = (Act\(_1\), Act\(_2\)). For any vertex \( v \in V \), and \( a \in \text{Act} \), let \( \delta(v, a) \) denote the \( a \)-th successor of \( v \). Games with \( |\text{Act}_i| > 2 \) can be equivalently modeled using two actions by adding intermediate states (see the remark after Definition 4).

We denote the finitely many \( \Xi_i \)-equivalence classes that matter (i.e. are not proper extensions of the leaves) by \( \mathcal{O}_i = \{O_j^i \mid j \in I\} \). An observation-based strategy then is, up to irrelevant moves, simply a function \( \sigma : \mathcal{O}^i \rightarrow \text{Act}_i \), which can be stored in linear space.

### 6.1 Characterization of Domination

We are going to give an algorithm to decide whether a given strategy is dominated in a finite tree game. To do so, we are going to simultaneously simulate \( \sigma \) and another strategy that is to be chosen by Player \( i \), in a product construction. We need the following additional definition:

> Definition 21 (Switching strategies). For any player \( i \), strategies \( \tau, \tau' \in \Sigma_i \) and an observation \( o \in O_i \), we denote \( \tau[o/\tau'] \) the strategy that plays \( \tau \) but upon visiting \( o \) switches to \( \tau' \). Formally,

\[
\tau[o/\tau'](o_1 \ldots o_n) = \begin{cases} 
\tau'(o_1 \ldots o_{i-1} o \ldots o_n) & \text{if } \exists i, o_i = o \\
\tau(o_1 \ldots o_n) & \text{otherwise}
\end{cases}
\]

Let us consider a finite tree game \( G = (V, E, V_1, V_2, \mathcal{O}_1, \mathcal{O}_2, \Omega_T) \) and let us note \( \Sigma_i \) the observation-based strategies of Player \( i \) in \( G \). For strategy \( \sigma \in \Sigma_i \) (\( G \)) and vertex \( v \in V_i \), let \( \delta^*(v, \sigma) \) be the vertex obtained by repeatedly applying \( \sigma \) as long as the vertex stays in \( V_i \). Thus, \( \delta^*_3(v) \) is either a leaf, or a vertex of \( V_{3-i} \). For \( v \notin V_i \), let \( \delta^*(v, \sigma) = v \). Given action \( a \) and strategy \( \sigma \), let us define \( a \cdot \sigma \) that plays \( a \) in the first step, and then switches to \( \sigma \).

> Definition 22. Given a finite tree game \( G(v_{\text{init}}) = (V, E, V_1, V_2, \mathcal{O}_1, \mathcal{O}_2, \Omega) \) with actions Act, and strategy \( \sigma \in \Sigma_i (G) \), define \( G_\sigma(v_{\text{init}}) = (V^\sigma, E^\sigma, V_1^\sigma, V_2^\sigma, \mathcal{O}_1^\sigma, \mathcal{O}_2^\sigma, \mathcal{O}_3^\sigma, \Omega^\sigma) \) with actions Act\(^\sigma\), where

- \( V^\sigma = V \times V \), \( V_1^\sigma = V \times V_1 \), and \( V_3^\sigma = V \setminus V_1 \),
- \( v_{\text{init}}^\sigma = (\delta^*(v_{\text{init}}, \sigma), v_{\text{init}}) \),
- \( \text{Act}_1^\sigma = \text{Act}_1 \), and \( \text{Act}_3^\sigma = \text{Act}_3 \),
- \( \mathcal{O}_1^\sigma = \{V \times \mathcal{O}_j^1 \mid j \in \mathcal{O}_1\} \), \( \mathcal{O}_3^\sigma = \{O_3^j \times \mathcal{O}_j^3 \mid O_j^3, O_3^j \in \mathcal{O}_3 \} \),
- For all \( (v, v') \in V_1^\sigma \), \( a \in \text{Act}_1^\sigma \),

\[
\delta^\sigma((v, v'), a) = (v, \delta(v', a)).
\]

For all \( (v, v') \in V_3^\sigma \), with \( \mathcal{O}_3^3(v) \neq \mathcal{O}_3^3(v') \), for all \( (a, b) \in \text{Act}_3^\sigma = \text{Act}_3 \),

\[
\delta_3^\sigma((v, v'), (a, b)) = (\delta^*(v, a \cdot \sigma), \delta(v', a)).
\]

For all \( (v, v') \in V_3^\sigma \), if \( \mathcal{O}_3^3(v) = \mathcal{O}_3^3(v') \), for all \( (a, b) \in \text{Act}_3^\sigma \),

\[
\delta^\sigma((v, v'), (a, b)) = \begin{cases} 
(\delta^*(v, a \cdot \sigma), \delta(v', b)) & \text{if } v, v' \notin \text{Leaves}(G), \\
(\delta^*(v, a \cdot \sigma), v') & \text{if } v, v' \notin \text{Leaves}(G), v' \in \text{Leaves}(G), \\
(v, \delta(v', b)) & \text{if } v \in \text{Leaves}(G), v' \notin \text{Leaves}(G),
\end{cases}
\]
Informally, Player $i$ only sees the second component and plays as in $G$; in fact, the second component of $G_\sigma$ reproduces precisely $G$. The first component always moves according to $\sigma$ from Player-$i$ vertices, and according to Player $3-i$'s actions otherwise. Player $3-i$ plays the same actions in both components as long as her observations match, but can choose different actions otherwise. Observe that we “accelerate” the transitions in the first component in Player-$i$ vertices, so we ensure that the first component is always either a Player $3-i$ vertex, or a leaf.

Note that $\Sigma_i^{\Sigma_i}(G) = \Sigma_i^{\Sigma_i}(G_\sigma)$ since the observations and actions for Player $i$ are identical in both games.

Consider game $G$, player $i$, and $\sigma, \sigma' \in \Sigma_i^{\Sigma_i}(G)$. For any strategy $\tau \in \Sigma_i^{\Sigma_i-1}(G)$, let us call $n_\tau^{\sigma,\sigma'}$ the smallest integer such that the observation of Player $3-i$ at the $n_\tau^{\sigma,\sigma'}$-th step is different between the plays $\rho = \iota(\text{Out}_G(\sigma, \tau))$ and $\rho' = \iota(\text{Out}_G(\sigma', \tau))$, that is, $n_\tau^{\sigma,\sigma'} = \min\{n | O^{3-i}(\rho_n) \neq O^{3-i}(\rho_n')\}$ when this minimum is finite, and $n_\tau^{\sigma,\sigma'} = \infty$ otherwise. Note that the lengths of the plays in the latter case must be equal since observations distinguish leaves from non-leaves. Let $\text{dist}_1^G(\sigma, \sigma') = \rho_{n_\tau^{\sigma,\sigma'}}$ and $\text{dist}_2^G(\sigma, \sigma') = \rho'_{n_\tau^{\sigma,\sigma'}}$ be these vertices where Player $3-i$ distinguishes both plays for the first time.

**Lemma 23.** Consider any game $G$, player $i$, $\sigma_1, \sigma_2 \in \Sigma_i^{\Sigma_i}(G)$, and $\tau \in \Sigma_i^{\Sigma_i-1}(G_{\sigma_1}).$ If we write $(t_1, t_2) = \ell(\iota(\text{Out}_{G_{\sigma_1}}(\sigma_2, \tau)))$, there exists $\tau \in \Sigma_i^{\Sigma_i-1}(G)$ such that $t_j = \ell(\iota(\text{Out}_G(\sigma_j, \tau)))$ for all $j \in \{1, 2\}$.

**Proof.** If $n_\tau^{\sigma_1,\sigma_2} = \infty$, then Player $3-i$ receives the same observation in both components. Let us define $\tau$ by $\tau(o) = a$ where $(a, b) = \tau((o, o))$. Then choosing $\tau_1 = \tau_2 = \tau$ yields the result.

Assume $n_\tau^{\sigma_1,\sigma_2} < \infty$, and write $(t_1, t_2) = \ell(\iota(\text{Out}_{G_{\sigma_1}}(\sigma_2, \tau)))$. We define $\tau(o)$ as the first component of $\tau((o, o))$ for all observations $o \in O_{3-i}$ such that $o \notin \{\text{dist}_1^G(\sigma_1, \sigma_2)\}_{j \in \{1, 2\}}$ or $o$ is not a descendant of these observations. Let $(s_1, s_2)$ be the $n_\tau^{\sigma_1,\sigma_2}$-th vertex of $\iota(\text{Out}_{G_{\sigma_1}}(\sigma_2, \tau))$. Notice that the run $\iota(\text{Out}_G(\sigma_j, \tau))$ visits $s_j$, for each $j \in \{1, 2\}$. For each $j$, in $\text{dist}_1^G(\sigma_1, \sigma_2)$, there exists strategy $\tau_j$ such that $\ell(\iota(\text{Out}_{G_{\sigma_j}}(\sigma_j, \tau_j))) = t_j$. We complete the definition of $\tau$ as follows: at each $\text{dist}_1^G(\sigma_1, \sigma_2)$, it switches to $\tau_j$.

Let us define $P_\sigma = \{(t, t') \in \text{Leaves}(G) \times \text{Leaves}(G) | t \in \Omega_t \Rightarrow t' \in \Omega_t\}$.

**Lemma 24.** For any $\sigma' \in \Sigma_i^{\Sigma_i}(G)$, $\sigma$ is weakly dominated by $\sigma'$ if, and only if $G_\sigma, \sigma' \models \text{AFP}_P$.

**Proof.** Assume $G_\sigma, \sigma' \models \text{AFP}_P$. Let $\tau \in \Sigma_i^{\Sigma_i-1}(G)$ such that $G, \sigma, \tau \models \Omega$; we will show that $G, \sigma', \tau \models \Omega$.

Let $\tau$ defined by $\tau((o, o')) = (\tau(o), \tau(o'))$ for all $o, o' \in O_{3-i}$. We have, by assumption, $G_\sigma, \sigma', \tau \models \text{F}P_\sigma.$ Moreover, the projection of this run to the first component is exactly $\iota(\text{Out}_G(\sigma, \tau)).$ The projection to the second component is $\iota(\text{Out}_G(\sigma', \tau)).$ By the definition of $P_\sigma$, and since $\text{Out}_G(\sigma, \tau)$ is in $\Omega$, $\text{Out}_G(\sigma', \tau)$ is in $\Omega$ too.

Conversely, assume that $G_\sigma, \sigma', \tau \models \text{G} \neg P_\sigma$. Let $(t, t') = \ell(\text{Out}(\sigma', \tau)G_\sigma)$, with $t \in \Omega_t$, $t' \notin \Omega_t$. Let $\tau \in \Sigma_i^{\Sigma_i-1}(G)$ be as given by Lemma 23. We get that $\iota(\text{Out}_G(\sigma, \tau)) = t$ and $\iota(\text{Out}_G(\sigma', \tau)) = t'$, which gives the desired result.

We let $Q_\sigma = \{(t, t') \in \text{Leaves}(G) \times \text{Leaves}(G) | t \notin \Omega_t \land t' \in \Omega_t\}$.

**Lemma 25.** For any $\sigma' \in \Sigma_i^{\Sigma_i}(G)$, $\sigma$ is dominated by $\sigma'$ if, and only if $G_\sigma, \sigma' \models \text{AFP}_P \land \text{EFQ}_\sigma$. 
<table>
<thead>
<tr>
<th>1 \ 2</th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>a</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 2 An admissible strategy of a sub-game may not be a part of an admissible strategy of the whole game.

**Proof.** Assume \( G, \sigma' \models \text{AFP}_i \land \text{EFQ}_i \). The previous lemma shows that \( \sigma' \) weakly dominates \( \sigma \). Let \( \tau \) such that \( G, \sigma', \tau \) ends in \((t, t') \in Q_\sigma \). Let \( \tau \) given by Lemma 23. We get that \( i(\text{Out}_G(\sigma, \tau)) = t \) and \( i(\text{Out}_G(\sigma', \tau)) = t' \), which means that \( \sigma \) is dominated by \( \sigma' \). ▷

**Lemma 26.** Given a game \( G \), player \( i \), and subsets \( P, Q \subseteq \text{Leaves}(G) \), one can decide in polynomial time if there is a strategy \( \sigma \in \Sigma_{i}^G \) such that \( G, \sigma \models \text{AFP}_i \land \text{EFQ}_i \).

**Proof.** We first show that there exist maximally permissive strategies for objectives of the type \( \text{AFP}_i \). In fact, define

\[
\text{Safe} = \{(o, a) \in O_i \times \text{Act}_i \mid \exists \sigma \in \Sigma_{i}^G, G, \sigma \models \text{AFP}_i \land \text{EFo}, \sigma(o) = a\}.
\]

This is the set of pairs of observation-actions \((o, a)\) such that some Player-\(i\) strategy that is winning for \( \text{AFP}_i \) is compatible with \( o \), and chooses \( a \) from \( o \). We claim that any \( \sigma \in \Sigma_{i}^G \) such that \( \forall o \in O_i, (o, \sigma(o)) \in \text{Safe} \) is winning for objective \( \text{AFP}_i \). This can be proved by induction on the length of plays, and using perfect recall.

Now, the set \( \text{Safe} \) can be computed bottom-up. Let \( G' \) be the game \( G \) where we only keep Player-\( i \) actions that conform to \( \text{Safe} \). There exists a strategy in \( G' \) winning for \( \text{AFP}_i \land \text{EFQ}_i \) if, and only if some vertex of \( Q \) is reachable in \( G' \).

▷ **Theorem 27.** Given game \( G \) on a finite tree, player \( i \) and strategy \( \sigma \in \Sigma_{i}^G \), one can decide in polynomial time whether \( \sigma \) is admissible.

### 6.2 Hardness

Let us start with a remark showing that one cannot hope to prune some actions of the game locally, to obtain a description of the admissible plays as it is the case in the perfect-information setting, see section on safety games in [12]. In fact, a strategy that only uses actions that appear in (the reachable part of) some admissible strategy may not be admissible itself.

Consider the game in normal form in Figure 2. It is easy to encode it as a game on a tree by ensuring that Player 2 is blind (i.e. all states in \( V_2 \) leads to the same observation) and players have perfect recall (in particular, they remember which actions they have played). Player 1 starts choosing \( a \) or \( b \), then Player 2 plays \((x, y \text{ or } z)\), and Player 1 plays again. We have that \( ab \) is admissible, so there is an admissible strategy playing \( a \) in the first vertex. Furthermore, in the subgame where Player 1 has already played \( a \), playing \( a \) is admissible (to see this, compare the first two lines in the figure). However strategy \( aa \) is dominated by \( ba \).

Similarly there is an admissible strategy of Player 1 playing \( b \) in the first step. Playing \( b \) is admissible in the subgame that starts with history \( b \), but the strategy \( bb \) is dominated by \( ab \).
We now show that deciding the existence of an admissible strategy choosing a particular action at a given state is NP-complete.

\textbf{Theorem 28.} Given two-player game $G$ on a finite tree, and an action $a$, deciding whether there is an admissible strategy $\sigma$ with $\sigma(x_{in}) = a$ is NP-complete.

\textbf{Proof.} NP-membership holds because strategies of Player 1 can be represented in polynomial size, and having guessed some strategy $\sigma_1$ which uses $a$, whether $\sigma_1$ is admissible can be verified in polynomial time by Corollary 27.

For NP-hardness, we encode an instance of the 3-SAT problem of the following form

$$\psi = \exists x_1, \ldots, x_n. C_1 \land \cdots \land C_m,$$

where $C_i = \ell_{i,1} \lor \ell_{i,2} \lor \ell_{i,3}$ and $\ell_{i,j} \in \{x_k, \neg x_k \mid k \in [1, n]\}$, for each $1 \leq i \leq m$, and $1 \leq j \leq 3$.

We define a game $G_\Omega$ as follows. From the initial vertex Player 1 either chooses a clause among $C_1, \ldots, C_m$, or goes to a special vertex $C_0$. Then Player 2 blindly selects a literal $\ell$ in $\{x_k, \neg x_k \mid k \in [1, n]\}$. At this point, Player 1 only sees the index $k$ such that $\ell \in \{x_k, \neg x_k\}$. Let us write $o^{i,k}$ for this observation, where $i$ is the index of the clause, or 0, and $k$ is the index of the variable chosen by Player 2.

From $o^{i,k}$, Player 1 chooses $T$ or $F$ meaning, intuitively, that $x_k$ is evaluated to true and false respectively. All other observations $o^{i,k}$ with $i > 0$ are leaves. The reachability condition $\Omega$ is satisfied when either Player 1 selects $C_i$ with $i > 0$ and $\ell$ is a literal which does not appear in $C_i$ or Player 1 selected $C_0$ and the valuation chosen in the last vertex is such that $\ell$ is evaluated to true.

Assume there is a strategy $\sigma_1$ choosing $C_0$ which is not dominated by any strategy not choosing $C_0$. Now, define the valuation $v$ as follows. For each variable $x_k$, we set $v(x_k)$ to true if, and only if, $\sigma_1(o^{0,k}) = T$. Let $\sigma'_i$ be any strategy of Player 1 choosing some clause $C_i$ with $i > 0$. Because $\sigma_1$ is not dominated by $\sigma'_i$, either: (A) there is a strategy $\tau$ of Player 2 which makes $\sigma_1$ win but not $\sigma'_i$; or (B) for all strategies $\tau$ of Player 2, $\sigma'_1, \tau$ wins implies $\sigma_1, \tau$ wins.

Let us first show that case (B) cannot happen. Recall that $\sigma'_1, \tau$ is winning when $\tau$ selects a literal outside $C_i$. If there are more than 2 variables ($n > 2$), which we can assume without loss of generality, then there is one variable $x_k$ and its negation that do not appear in $C_i$. Consider the strategies $\tau$ and $\tau'$ that select $x_k$ and $\neg x_k$ respectively. Since they make $\sigma'_1$ win, they must also make $\sigma_1$ win, which means that the valuation defined by $\sigma_1$ makes both $x_k$ and $\neg x_k$ true, which is a contradiction.

Thus, only the case (A) is possible. For each $1 \leq i \leq m$, let $\ell_i$ be the strategy that chooses $C_i$. Let $\tau'$ be a Player-2 strategy such that $\sigma_1, \tau'$ wins and $\sigma'_i, \tau'$ loses. Let us fix $1 \leq i \leq m$, and let $\ell$ denote the literal chosen by $\tau'$. The run of $\sigma'_1, \tau'$ is losing, which means that $\ell \in C_i$. Since $\sigma_1, \tau'$ wins, $\ell$ is satisfied by $v$. Since $\ell \in C_i$, $C_i$ is satisfied by $v$. Since $i$ was arbitrary, all clauses $C_i$ are satisfied. Hence $\psi$ is satisfiable.

In the other direction, assume the formula is satisfiable. Let $v$ be a valuation satisfying the formula, and $\sigma_0$ be the strategy that first plays $C_0$, and then plays according to the valuation $v$. Let $\sigma'_i$ be any strategy that first plays $C_i$ for any $i \in [1, n]$. As $v$ satisfies the formula, there is a literal $\ell$ in $C_i$ such that $v(\ell) = T$. Let $\tau_\ell$ be the strategy of Player 2 that chooses this literal. As $\ell \in C_i$, $\tau_\ell$ makes $\sigma'_i$ lose, and as $v(\ell) = T$, $\tau_\ell$ makes $\sigma_1$ win. So $\sigma_1$ is not dominated by $\sigma'_i$. ◀
In this section, we study decision problems related to admissible strategies for regular infinite sequential games with imperfect information. We have seen in Theorem 17 that a strategy of Player 1 that is dominated when Player 2 is perfectly informed is also dominated when Player 2 is imperfectly informed. The special case of a perfectly informed Player 2 is thus important. In this section, we start by considering this case, exploiting the characterization obtained in Theorem 20. Then we turn to the case when Player 2 is not perfectly informed. Before presenting our results, we need to recall some notions related to infinite trees and automata on infinite trees.

### 7.1 Automata on infinite trees and tree encodings of strategies

**Infinite trees**

Following e.g. [21], given a finite set \( D \), a \( D \)-tree is a prefix closed language \( T \subseteq D^* \). When \( D \) is clear from the context, we call \( T \) a tree. Elements of \( T \) are called nodes, and the empty word \( \epsilon \) is the root of the tree. For every \( x \in T \), the nodes \( x \cdot d \in T \) for \( d \in D \) are the children of \( x \), the children \( x \cdot d \) is called the child of direction \( d \) of \( x \). A \( D \)-tree is a full infinite tree if \( T = D^* \). A branch \( b \) of \( T \) is a set \( b \subseteq T \) such that \( \epsilon \in b \) and for every \( x \in b \) there exists a unique \( d \in D \) such that \( x \cdot d \in b \). Given two sets \( D \) and \( L \), a \( L \)-labelled \( D \)-tree is a pair \( \langle T, L \rangle \) where \( T \) is a \( D \)-tree and \( L : T \rightarrow L \) labels each node of \( T \) with an element of \( L \).

**Tree encoding of observation-based strategies**

We encode observation-based strategies \( \sigma \in \Sigma_1^{\infty} \), i.e. \( \sigma : (O^1) * \cdot O^1_{\text{turn}} \rightarrow \{0, 1\} \), as full infinite \( \{0, 1, *\} \)-labelled \( O^1 \)-trees with the convention that:

- the root node is always labelled with *;
- every node that follows an observation \( o \in O^1_{\text{turn}} \) is labeled with 0 or 1 according to \( \sigma \);
  - if the node is reached with a sequence of directions \( \rho = o_1 o_2 \ldots o_n \) that corresponds to observations in \( O^1 \) and \( o_n \in O^1_{\text{turn}} \), then the node is labelled by \( \sigma(\rho) \);
- every node that follows an observation \( o \in O^1 \setminus O^1_{\text{turn}} \) is labelled with * as in that case, it is the turn of Player 2 to play and the choice of Player 2 is not visible for Player 1.

**Alternating tree automata**

We use alternating tree automata (AT) to recognize regular sets of infinite trees that encode observation-based strategies. A AT \( P = (Q, q_0, \Delta, \Psi) \) that operates on \( L \)-labelled \( D \)-trees is defined by:

- its finite non-empty set of states \( Q \);
- its initial state \( q_0 \);
- its transition function \( \Delta : Q \times L \rightarrow B^+(Q \times D) \), where \( B^+(Q \times D) \) is the set of (positive) Boolean formula built from elements in \( Q \times D \) using \( \vee \) and \( \wedge \). For a set \( E \subseteq Q \times D \) and a formula \( \psi \in B^+(Q \times D) \), we say that \( E \) satisfies \( \psi \) iff assigning true to elements in \( E \) and assigning false to elements in \( (Q \times D) \setminus E \) makes \( \psi \) true;
- \( \Psi \) is the acceptance condition which is a set of infinite sequences of states in \( Q \) that are accepting. Typically, we consider either parity functions, giving alternating parity tree automata (APT), or Muller conditions (that subsume Boolean combinations of parity conditions), giving alternating Muller tree automata (AMT), to define \( \Psi \).
A run of $P = (Q, q_0, \Delta, \Psi)$ on a $\mathcal{L}$-labelled $\mathcal{D}$-tree $(T, L)$ is $Q \times \mathcal{D}^*$-labelled tree $(T_r, r)$ such that:
1. $\epsilon \in T_r$ and $r(\epsilon) = (q_0, \epsilon)$;
2. Let $y \in T_r$ with $r(y) = (q, x)$ and $\Delta(y, L(x)) = \psi$. Then there exists a (possibly empty) set $E = \{(q_0, d_0), (q_1, d_1), \ldots, (q_n, d_n)\} \subseteq Q \times D$ and:
   a. $E$ satisfies $\psi$;
   b. for all $i$, $0 \leq i \leq n$, we have $y \cdot i \in T_r$ and $r(y \cdot i) = (q_i, x \cdot d_i)$.
A run is accepting iff all its infinite branches $b^\infty$ are such that $b^\infty \in \Psi$, i.e. they satisfy the acceptance condition.

### 7.2 Player 2 perfectly informed

As we mentioned following Theorem 20, in regular games the validity of the condition in the theorem depends only on the set of last vertices in the histories forming a monochromatic set. In slight abuse of notation, we call a set of vertices arising in this way a monochromatic set, too.

**Lemma 29.** Let $M = M_w \sqcup M_c \sqcup M_\ell$ be a monochromatic set. We can construct in PTIME an APT that recognizes the set of trees that encode observation-based strategies $\sigma : (O^1)^* \cdot O^{\text{turn}}_1 \to \{0, 1\}$ of Player 1 and such that $M_w = M^w_1$, $M_c = M^c_1$, and $M_\ell = M^\ell_1$.

**Proof.** The main idea of the construction is as follows. The APT has four parts:
1. one that checks that $\sigma$ ensures a win from all vertices in $V_w$ no matter what are the choices made by Player 2;
2. one that checks that $\sigma$ can cooperate with Player 2 to win from all vertices in $V_c$;
3. one that checks that $\sigma$ can cooperate with Player 2 to lose from all the vertices in $V_c$;
4. one that checks that all outcomes that are compatible with $\sigma$ are losing from all the vertices in $V_\ell$ no matter what are the choices made by Player 2.

Accordingly, its state space is defined as $Q = \{q_0\} \cup V \times \{w, c_w, c_\ell, \ell\}$. The transitions are defined as follows:

- for states in $V \times \{w\}$. If a node $n$ of the tree is reached by a run of the APT that ends up in state $(v, w)$ then we must verify that the strategy encoded in the tree ensures to win no matter what are the choices of Player 2. So, the automaton follows the choices prescribed by the strategy for nodes annotated by $i \in \{0, 1\}$ and branches universally on the choices of Player 2 for nodes annotated by *. Accordingly:
  - for $i \in \{0, 1\}$: $\Delta((v, w), i) = ((s(v), i), w), O^1(s(v), i))$
  - for *: $\Delta((v, w), *) = ((s(v, 0), w), O^1(s(v, 0))) \land ((s(v, 1), w), O^1(s(v, 1)))$

- for states in $V \times \{c_w\}$. If a node $n$ of the tree is reached by a run of the APT that ends up in state $(v, c_w)$ then the automaton must verify that the strategy $\sigma$ encoded in the tree ensures that at least one outcome is winning. So, the automaton follows the choices of Player 1 as encoded in the tree and for nodes labelled with * that denote choices of Player 2, the automaton non-deterministically chooses one choice for Player 2. Accordingly:
  - for $i \in \{0, 1\}$: $\Delta((v, c_w), i) = ((s(v, i), c_w), O^1(s(v, i)))$
  - for *: $\Delta((v, c_w), *) = ((s(v, 0), c_w), O^1(s(v, 0))) \lor ((s(v, 1), c_w), O^1(s(v, 1)))$

- for states in $V \times \{c_\ell\}$. The approach is similar. If a node $n$ of the tree is reached by a run of the APT that ends up in state $(v, c_\ell)$ then the automaton must verify that the strategy $\sigma$ encoded in the tree ensures that at least one outcome compatible with $\sigma$ is losing. So, the automaton follows the choices of Player 1 as encoded in the tree and for nodes labelled with * that denote choices of Player 2, the automaton non-deterministically
chooses one choice for Player 2. The difference with the previous case \((c_w)\) is handled by the acceptance condition. Accordingly:

- for \(i \in \{0, 1\}\): \(\Delta((v, c_l), i) = ((s(v, i), c_l), O^1(s(v, i)))\)
- for \(*\): \(\Delta((v, c_l), *) = ((s(v, 0), c_l), O^1(s(v, 0))) \vee ((s(v, 1), c_l), O^1(s(v, 1)))\)
- for states in \(V \times \{\ell\}\). If a node \(n\) of the tree is reached by a run of the APT that ends up in state \((v, \ell)\) then we must verify that the strategy \(\sigma\) encoded in the tree cannot win no matter what Player 2 chooses. So, the automaton follows the choices prescribed by the strategy \(\sigma\) for nodes annotated by \(i \in \{0, 1\}\) and branches universally on the choices of Player 2 for nodes annotated by \(*\). The difference with the first case \((w)\) is handled by the acceptance condition. Accordingly:

- for \(i \in \{0, 1\}\): \(\Delta((v, \ell), i) = ((s(v, i), \ell), O^1(s(v, i)))\)
- for \(*\): \(\Delta((v, \ell), *) = ((s(v, 0), \ell), O^1(s(v, 0))) \land ((s(v, 1), \ell), O^1(s(v, 1)))\)

It remains now to define the acceptance condition. First, we remark that a run of the APT directly jump from its initial state \(q_0\) to one of the four parts whose transitions have been described above. When entering one of those parts, the run will stay in that part of the state space for ever. The runs that have have entered the part associated to \(w\) and to \(c_w\) must simulate paths in the game graph that are in \(W\), those that are in the part associated to \(c_l\) and to \(\ell\) must simulate paths that are outside \(W\). \(W\) is defined by a parity condition \(pr\). We thus define the party condition \(pr'\) of our automaton as follows: for all \(v \in V\), \(pr'(v, w) = pr'(v, c_w) = pr(v)\) (the parity condition is preserved) and \(pr'(v, c_l) = pr'(v, \ell) = pr(v) + 1\) (the parity condition is inverted).

As the emptiness problem for APT is solvable in \(\text{ExpTime}\), we deduce the following corollary:

**Corollary 30.** Given a partition \(M = M_w \sqcup M_c \sqcup M_\ell\) of a monochromatic set, we can decide in \(\text{ExpTime}\) if there exists an observation-based strategy \(\sigma: (O^1)^* \cdot O^1_{\text{turn}} \rightarrow \{0, 1\}\) that induces this partition.

**Lemma 31.** We can construct in \(\text{ExpTime}\) an APT \(P\) that accepts the \(\{0, 1, *\}\)-labelled \(O^1\)-trees that are the tree encodings of the observation-based strategies \(\sigma \in \Sigma^1_1\) that are dominated in \(R^U(v_0)\) where Player 2 is perfectly informed.

**Proof.** Let \(R(v_0)\) be a regular game. First, we note that Lemma 29 allows us to compute, for all monochromatic sets \(K\), all the partitions \(K_w \sqcup K_c \sqcup K_\ell\) that are witnessed by observation-based strategies of Player 1. Among those, we can extract all the partitions \(K_w \sqcup K_c \sqcup K_\ell\) that correspond to strategies that are dominated following Theorem 20. There are at most an exponential number of them in the size of the regular game. We note \(B\) all those partitions.

Our APT will first guess a finite observation history \(\rho\) and compute the knowledge associated to \(\rho\) while following the strategy \(\sigma\) encoded in the tree. Let \(K\) be the resulting monochromatic set of vertices. This set is stored in the state of the APT and then the APT nondeterministically chooses a partition \(K_w \sqcup K_c \sqcup K_\ell\) of \(K\) in \(B\). Then the APT verifies that the strategy in the tree induces that partition. This is done as in the proof of Lemma 29.

As APT are closed under all Boolean operations, see e.g. [19], the set of admissible observation-based strategies of Player 1 is effectively omega-regular:

**Theorem 32.** When Player 2 is perfectly informed, the set of admissible observation-based strategies of Player 1 in \(R^U(v_0)\) is effectively \(\omega\)-regular and the emptiness of this set is decidable in \(2\text{ExpTime}\).
7.3 Player 2 imperfectly informed

The hierarchical case

We start by considering the case in which both players have imperfect information but Player 2 is more informed than Player 1. In this case, the informedness of players is hierarchical in the sense of [7]. For the hierarchical case, we can decide if the set of admissible observation-based strategies of Player 1 is empty or not. We obtain this result by a reduction to the model-checking problem of Strategy Logic with Imperfect Information [5], SLii, for short.

We only recall here informally the syntax and semantics of SLii formulas and refer the interested reader to [5] for formal definitions. We start with the case where the players are perfectly informed. Strategy Logic (SL) extends the linear temporal logic (LTL) [3] and treats strategies $x$ as first-order objects that can be quantified: $\langle\phi\rangle$ reads "there exists a strategy $x'$ while $[x]$ reads 'for all strategies $x$'. In SL, strategies can be bound to players: $(x, 1)$ reads 'Player 1 uses strategy $x$'. As an example, let $\phi$ be an LTL formula, and consider the following SL formula $\langle\phi\rangle[y](y, 1)(y, 2)\phi$. This formula expresses that there exists a strategy $x$ for Player 1 such that for all strategies $y$ of Player 2, when the two players play their respective strategies $x$ and $y$ then the outcome from the initial vertex of the game arena is a path that satisfies $\phi$. So, this formula expresses the existence of a winning strategy for Player 1 in a two-player zero sum game with objective $\phi$ for Player 1 and $\neg\phi$ for Player 2. SL can express many interesting game properties such as the existence of Nash equilibria, the existence of dominating strategies, etc., see [14, 22] for more examples.

When strategy logic is interpreted over a game in which players have imperfect information, then the strategy quantifier explicitly limits the quantification to observation-based strategies: $\langle\phi\rangle^O \text{ reads 'there exists an } O\text{-observation-based strategy } x'$. However, to obtain the decidability of the model-checking problem for SLii, quantifiers must respect constraints that ensure 'hierarchical instances'.

Intuitively, a formula of SLii is hierarchical if, as one goes down the syntactic tree of the formula, the observations annotating strategy quantifications can only become finer. So, for example, if in the syntactic tree, quantification $\langle\phi\rangle^{O_1}$ is followed by $\langle\phi\rangle^{O_2}$ then it must be the case that $O_2$ is finer than $O_1$ meaning that for all $v_1, v_2 \in V$, if $O_2$ does not distinguish $v_1$ and $v_2$, i.e. $O_2(v_1) = O_2(v_2)$, then it is also the case for $O_1$, i.e. $O_1(v_1) = O_1(v_2)$.

Let us now consider a regular game $R(v_0) = (V, E, V_1, V_2, O_1, O_2)$ such that $O_2$ is finer than $O_1$ that induces the infinite sequential game with imperfect information $R^U(v_0) = (d, W, \equiv_1, \equiv_2)$. Let $\phi_1$ be a LTL specification for the objective of Player 1 then the following SLii is hierarchical and evaluates to true in $v_0$, if and only if, Player 1 has admissible $O_1$-observation-based strategies when Player 2 plays $O_2$-observation-based strategies:

$$
\langle\phi\rangle^{O_1} [x']^{O_1} \langle\phi\rangle^{O_2}(x, 1)(y, 2)\phi_1 \land (x', 1)(y, 2)\neg\phi_1 \\
\lor [y]^{O_2}(x', 1)(y, 2)\phi_1 \rightarrow (x, 1)(y, 2)\phi_1
$$

\[\blacktriangleright\text{Theorem 33. Let } R(v_0) = (V, E, V_1, V_2, O_1, O_2) \text{ be a regular game such that } O_2 \text{ is finer than } O_1 \text{ and that induces } R^U(v_0) = (d, W, \equiv_1, \equiv_2). \text{ The existence of admissible strategies for Player 1 in } R^U(v_0) \text{ is decidable.}\]

The general case

This case is more involved and so far we did not succeed to obtain a general characterization of dominated strategies in the form of Theorem 20. Nevertheless, we are able to characterize all the observation-based strategies of Player 1 that dominate a regular observation-based
strategy given as a finite automaton $A$ (as in Definition 6). Our characterization is effective: we can construct from a regular game $R$ and a finite automaton $A$, an AMT $P$ that accepts the tree encodings of all the observation-based strategies $\sigma$ that dominate the regular strategy $\sigma_A$ defined $A$. The construction of the AMT is given in the proof of the following lemma:

Lemma 34. Let $R(v_0)$ be a regular game that induces $R^2(v_0) = (d, W, \equiv_1, \equiv_2)$, and let $A$ be a finite automaton that encodes a regular observation-based strategy $\sigma_A \in \Sigma^\equiv_1$. We can construct in PTime an AMT $P$ that accepts all the tree encodings of observation-based strategies $\sigma \in \Sigma^\equiv_2$ that dominates $\sigma_A$ when Player 2 plays strategies in $\Sigma^\equiv_2$.

Proof. Remember that the strategy $\sigma_A$ is dominated by a strategy $\sigma$ when Player 2 plays observation-based strategies if and only if:

1. $\forall \pi \in \Sigma^\equiv_2_1 : \text{Out}(\sigma_A, \pi) \in W \rightarrow \text{Out}(\sigma, \pi) \in W$
2. $\exists \pi \in \Sigma^\equiv_2_2 : \text{Out}(\sigma_A, \pi) \notin W \land \text{Out}(\sigma, \pi) \in W$

Here the strategy $\sigma_A$ is encoded by the automaton $A = (Q^A, q^A_0, O^1, \delta^A, F^A)$, and the strategy $\sigma$ is the strategy encoded in the tree. We use a AMT to check those two properties: its state space is structured in two disjunct parts plus an initial state $q_0$: $Q = \{q_0\} \cup Q^1 \cup Q^2$. States in $Q^1$ are used to check the first property, and in $Q^2$ the second property. We now give a detailed description of those sets of states: $Q^i = \{i\} \times V \times V \times \{0, 1\} \times Q^A$, $i \in \{1, 2\}$.

Intuitively, a state $(i, v_1, v_2, \text{dist}, q)$ is reached when:

- if $i = 1$, the ATP is checking the first condition, and if $i = 2$, the AMT is checking the second condition;
- the interaction of the strategy $\pi$ played by Player 2 against $\sigma_A$ leads to vertex $v_1$;
- the interaction of the strategy $\pi$ played by Player 2 against $\sigma$ leads to vertex $v_2$;
- $\text{dist}$ is true if and only if Player 2 has been able to distinguish, based on its observation $O^2$, between the history generated by strategy $\sigma$ against $\pi$ and the history generated by the strategy $\sigma_A$ against $\pi$, and false otherwise;
- $q$ is the state reached by the automaton $A$, that encodes $\sigma_A$ on the current sequence of $O^1$-observations.

In the part $Q^1$ of the AMT, the automaton branches universally on the choices of Player 2 in order to consider all possible strategies $\pi$ that Player 2 can play and it verifies that, for all of them, if the outcome against $\sigma_A$ is winning then the outcome against $\sigma$ is also winning. In the part $Q^2$ of the AMT, the automaton branches existentially on the choices of Player 2 in order to guess a strategy of Player 2 that forces an outcome in the complement of $W$ against $\sigma_A$ and in $W$ against $\sigma$. This is realized by the following transition relation and acceptance condition:

Transition function. Let $(i, v_1, v_2, \text{dist}, q^A)$, and we distinguish between the universal $(i = 1)$ and existential $(i = 2)$ cases:

- in the universal part:
  - if the label of the current node in the tree is $\ast$ then it is the turn of Player 2 to play. Then either Player 2 has already distinguished the histories that ends up in $v_1$ and $v_2$ respectively, i.e. $\text{dist} = 1$ and so he can possibly play differently in the two cases:
    \[
    \Delta((1, v_1, v_2, 1, q^A), *) = \\
    \bigwedge_{j_1, j_2 \in \{0, 1\}} (\{(1, s(v_1, j_1), s(v_2, j_2), 1, \delta^A(q^A, O^1(s(v_1, j_1)))), O^1(s(v_2, j_2)))
    \]
or Player 2 has not yet distinguished the histories that ends up in $v_1$ and $v_2$ respectively, i.e. $\text{dist} = 0$, and so Player 2 makes the same choices in the two branches:

$$
\Delta((1, v_1, v_2, 0, q^A), *) \\
= \\
\bigvee_{j_1,j_2 \in \{0,1\}} \left( (1, s(v_1, j_1), s(v_2, j_2), \text{dist'}(j), \delta^A(q^A, O^1(s(v_1, j_1))), O^1(s(v_2, j_2))) \right)
$$

where $\text{dist'}(j) = (O^2(s(v_1, j)) \neq O^2(s(v_2, j)))$, i.e. the new value of $\text{dist}$ is set to 1 only if Player 2 received two different observations for $s(v_1, j)$ and $s(v_2, j)$, in that case, he can now distinguish between the two branches.

- If the label of the current node in the tree is $c \in \{0,1\}$ then it is the turn of Player 1 to play and $c$ is the choice made by $\sigma$, while the choice $d$ made by $\sigma_A$ is equal to 0 if $q^A \in F^A$ otherwise it is equal to 1:

$$
\Delta((1, v_1, v_2, \text{dist}, q^A), c) \\
= \\
(1, s(v_1, d), s(v_2, c), \text{dist'}(j), \delta^A(q^A, O^1(s(v_1, d))), O^1(s(v_2, c)))
$$

where $\text{dist'}(j) = \text{dist} \lor (O^2(s(v_1, d)) \neq O^2(s(v_2, c)))$, i.e. the two histories are distinguishable if they were before or if Player 2 gets two different observations in this round.

- in the existential part:

  - If the label of the current node in the tree is $*$ then it is the turn of Player 2 to play. Then either Player 2 has already distinguished the histories that ends up in $v_1$ and $v_2$ respectively, i.e. $\text{dist} = 1$:

$$
\Delta((1, v_1, v_2, 1, q^A), *) \\
= \\
\bigvee_{j_1,j_2 \in \{0,1\}} \left( (1, s(v_1, j_1), s(v_2, j_2), 1, \delta^A(q^A, O^1(s(v_1, j_1))), O^1(s(v_2, j_2))) \right)
$$

or Player 2 has not already distinguished the histories that ends up in $v_1$ and $v_2$ respectively, i.e. $\text{dist} = 0$, so Player 2 makes the same choices in the two branches:

$$
\Delta((1, v_1, v_2, 0, q^A), *) \\
= \\
\bigvee_{j \in \{0,1\}} \left( (1, s(v_1, j), s(v_2, j), \text{dist'}(j), \delta^A(q^A, O^1(s(v_1, j))), O^1(s(v_2, j))) \right)
$$

where $\text{dist'}(j) = (O^2(s(v_1, j)) \neq O^2(s(v_2, j)))$, i.e. the new value of $\text{dist}$ becomes equal to 1 only if Player 2 received two different observations for $s(v_1, j)$ and $s(v_2, j)$, if so, he can now distinguish between the two branches. Note that those formulas are the same as the one for the universal part but with the conjunction replaced by a disjunction.

- If the label of the current node in the tree is $d \in \{0,1\}$ then it is the turn of Player 1 to play and $d$ is the choice made by $\sigma$, while the choice $c$ made by $\sigma_A$ is equal to 0 if $q^A \in F^A$ otherwise it is equal to 1, and then we have that:

$$
\Delta((1, v_1, v_2, 1, q^A), c) \\
= \\
(1, s(v_1, c), s(v_2, d), \text{dist'}(j), \delta^A(q^A, O^1(s(v_1, c))), O^1(s(v_2, d)))
$$

where $\text{dist'}(j) = \text{dist} \lor (O^2(s(v_1, c)) \neq O^2(s(v_2, d)))$, i.e. the two histories are distinguishable if they were before or if Player 2 gets two different observations now.
Acceptance condition. Again, we distinguish between the universal and existential parts:

- Let \( r \) be a run of the AMT in the universal part and let \( \text{inf}(r) \) be the set of states that repeats infinitely often along \( r \). Let \( P_r \) be the set of pairs of vertices \((v_1, v_2)\) that appear in states of the automaton in \( \text{inf}(r) \), i.e. pairs of vertices that appear infinitely often along the run \( r \). We declare the set of states \( \text{inf}(r) \) to be accepting when we verify that: if \( \min(v_1, v_2) \in P_r \) then \( p(v_1) \) is even, i.e. a run is good if the run simulates an execution in which Player 2 plays a strategy \( \pi \), and if the outcome of \( \sigma_A \) and \( \pi \) is winning for the parity condition \( pr \), then it is also the case for the outcome of \( \sigma \) and \( \pi \). This is a valid Muller condition.

- Let \( r \) be a run of the AMT in the existential part and let \( \text{inf}(r) \) be the set of states that repeats infinitely often along \( r \). Let \( P_r \) be the set of pairs of vertices \((v_1, v_2)\) that appear in states of the automaton in \( \text{inf}(r) \), i.e. pairs of vertices that appear infinitely often along the run \( r \). We declare the set of states \( \text{inf}(r) \) to be accepting when we verify that: if \( \min(v_1, v_2) \in P_r \) then \( p(v_2) \) is even, i.e. a run is good if the run simulates an execution in which Player 2 plays a strategy \( \pi \), and if the outcome of \( \sigma_A \) and \( \pi \) is losing for the parity condition \( pr \), and the outcome of \( \sigma \) and \( \pi \) is winning for the parity condition \( pr \). This is a valid Muller condition.

As a consequence, we obtain:

**Theorem 35.** Let \( R(v_0) \) be a regular game that induces \( R^U(v_0) = (d, W, \Sigma_1, \Sigma_2) \), and let \( A \) be a finite automaton that encodes a regular observation-based strategy \( \sigma_A \in \Sigma_1 \). The problem to decide if \( \sigma_A \) is dominated when Player 2 plays strategies in \( \Sigma_2^\infty \) is \( \text{ExpTime-C} \).

Proof. We can solve the problem in \( \text{ExpTime} \) thanks to the AMT construction of polynomial size given in lemma 34 and the fact that emptiness of AMT is solvable in \( \text{ExpTime} \). For hardness, it is easy to reduce the problem of deciding the winner in a zero-sum reachability game with imperfect information to our problem, and this problem is complete for \( \text{ExpTime} \) [13].

Acknowledgements. We would like to thank Marie Van den Bogaard for carefully reading and commenting on a previous version of this paper.

References

Admissibility in Games with Imperfect Information


Probabilistic Programming

Hongseok Yang

University of Oxford, UK
hongseok00@gmail.com

Abstract

Probabilistic programming refers to the idea of using standard programming constructs for specifying probabilistic models from machine learning and statistics, and employing generic inference algorithms for answering various queries on these models, such as posterior inference and estimation of model evidence. Although this idea itself is not new and was, in fact, explored by several programming-language and statistics researchers in the early 2000, it is only in the last few years that probabilistic programming has gained a large amount of attention among researchers in machine learning and programming languages, and that expressive and efficient probabilistic programming systems (such as Anglican, Church, Figaro, Infer.net, PSI, PyMC, Stan, and Venture) started to appear and have been taken up by a nontrivial number of users.

The primary goal of my talk is to introduce probabilistic programming to the CONCUR/QUEST/FORMATS audience. At the end of my talk, I want the audience to understand basic results and techniques in probabilistic programming and to feel that these results and techniques are relevant or at least related to what she or he studies, although they typically come from foreign research areas, such as machine learning and statistics. My talk will contain both technical materials and lessons that I learnt from my machine-learning colleagues in Oxford, who are developing a highly-expressive higher-order probabilistic programming language, called Anglican. It will also include my work on the denotational semantics of higher-order probabilistic programming languages and their inference algorithms, which are jointly pursued with colleagues in Cambridge, Edinburgh, Oxford and Tubingen.

1998 ACM Subject Classification D.3.1 Formal Definitions and Theory, G.3 Probability and Statistics

Keywords and phrases Probabilistic programming, Machine learning, Denotational semantics

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2017.3

Category Invited Talk
A New Notion of Compositionality for Concurrent Program Proofs

Azadeh Farzan\textsuperscript{1} and Zachary Kincaid\textsuperscript{2}

\textsuperscript{1} University of Toronto, Ontario, Canada  
azadeh@cs.toronto.edu  
\textsuperscript{2} Princeton University, NJ, USA  
zkincaid@cs.princeton.edu

Abstract

This paper presents a high level overview of Proof Spaces \cite{Farzan:2017:PLC} as an instance of a new approach to compositional verification of concurrent programs and discusses potential future work extending the approach beyond its current scope of applicability.

1998 ACM Subject Classification D.2.4 Software/Program Verification

Keywords and phrases Concurrency, Proofs, Dynamic Memory, Recursion

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2017.4

Category Invited Talk

1 Verification of Parameterized Concurrent Programs

Compositional proofs have always been the holy grail of reasoning about concurrent programs. Two seminal contributions, namely Owicki-Gries style \cite{Gries:1976:OOG} and rely-guarantee \cite{Dimitrijevic:1989:RGL}, have set the tone for compositional proofs of concurrent programs and inspired decades of research in this area. In both techniques (and the wealth of follow-up work), compositionality strictly means composing correctness arguments over individual threads into a correctness argument for the entire program, with some extra supporting arguments that are preferably kept at a minimum required. In a recent line of work \cite{Farzan:2017:PLC,Farzan:2017:PCF,Farzan:2017:RDF,Farzan:2017:PGP}, we proposed a new approach to static analysis and verification of concurrent programs which include an unbounded number of concurrent threads with local and global memory, with a different notion of compositionality. This talk will provide an overview of this approach, which takes a different view of compositionality. In this paper, we hope to look forward and through examples discuss early evidence of why we believe our approach to be a promising framework for future research into extensions of the program models (of the already published work \cite{Farzan:2017:PLC,Farzan:2017:PCF,Farzan:2017:RDF,Farzan:2017:PGP}) into more expressive models including features like dynamic memory and recursion.

We will do an informal exposition of our approach from \cite{Farzan:2017:PLC} using a simple example. The interested reader can refer to \cite{Farzan:2017:PLC,Farzan:2017:PCF,Farzan:2017:RDF,Farzan:2017:PGP} for the technical details. The principle behind our methodology is simple. It is difficult to reason about the correctness of a complex concurrent program, however, it is much simpler to reason about the correctness of a single behaviour (i.e. a single run) of the same program. Consider, for example, the complexity of reasoning about unbounded concurrency. Any (terminating) run of such a program always includes a bounded (by the length of the run) number of participating threads. Therefore, the complexity of dealing with unboundedly many threads can be circumvented whilst reasoning about the run. The idea is then to mine these simple proofs of program runs for ingredients to construct a correctness proof for the program.
This effectively introduces a different way of decomposing the correctness argument for a concurrent program. The new proof is a non-trivial composition of proofs of correctness constructed for finitely many runs of the program. The correctness proof for each run represents a class of behaviours for the program with a common correctness argument. Instead of being forced to decompose the proof argument using the program syntax as a strict guide, one can observe how the program behaves in different scenarios and reason that in each case it exhibits the correct behaviour. Our constructed proofs may end up with structures very similar to or wildly different from the original program. This is extra flexibility is useful specially when a compositional proof in the classic sense is either nonexistent or hard to automatically construct.

The diagram in Figure 1 illustrates our methodology. The main goal of this methodology is to introduce a clean separation between

- logical reasoning about data, that is over domains of the program variables, and
- purely combinatorial reasoning about sufficiency of a candidate proof for a given program.

In many settings, distinctly so for concurrent programs, an argument for correctness often involves both types of reasonings. The thesis of this methodology is that this separation could make each argument type simpler, and moreover, each side of the argument becomes amenable to existing technology. In particular, on the logical reasoning side we can make use of the wealth of techniques that have been developed for loop invariant generation, such as abstract interpretation [6, 7], Craig interpolation [15, 18, 2], and constraint-based techniques [5, 14]. On the combinatorial side, we can make use of technology behind finite-state model checking, such as partial order reduction [32, 23, 13] and symbolic state-space exploration [4, 17].

The key steps in this methodology are:

- **Proof generation for a run.** Depending on the richness of the program model, this task can be straightforward with an array of existing solutions, or more tricky where new algorithms need to be proposed for it. In particular:
  - In [9], program runs are simple (straight line sequential) programs with integer typed variables, where constructing a proof for a run is a well-understood task. Since concurrent programs are assumed to have a fixed number of threads, the challenge is the well-known state explosion problem, and therefore the goal is compactness of proofs. The innovation of proof construction step in this case is focused on constructing parallel in proofs in form of inductive data flow graphs, that would generalize the proof of correctness of a run to many other similar runs.
  - In [10], where the program model was extended to include unboundedly many threads, a special type of proof, namely a counting proof, is constructed for a run. A special
 technique for proof generation is proposed that automatically constructs a counting argument (i.e. a proof making use of counters) for a run.

- In [11], program runs are syntactically identical to those in [9], however, these program runs are (terminating) runs for a program with unboundedly many threads. A proof generated for a run in this context needs to be generalizable to the proof for a set of runs for such a program, and therefore, a different notion of proof, namely a proof space was introduced for these programs, a special proof generalization symmetry rule to tackle unbounded concurrency (more on this later).

- In [12], the program model is identical to the previous item, but the runs can potentially be non-terminating, since proving termination (and liveness in general) is the goals. Proofs for such runs, therefore, need to include termination arguments, which were ranking functions in this case.

- **Proof checking.** This is an algorithmic step that needs to (i) check if the candidate proof is indeed a proof of the given property for the given program, and (ii) provide a counter example if it is not, to ensure progress in the overall proof construction scheme (i.e. the choice of the run \( \tau \) in (a) in Figure 1). This requires effective representations of both the program and the candidate proof so that sufficiency of the proof can be algorithmically checked. We used alternating finite automata in [9], Petri nets in [10], and a new notion of data automata in [11, 12] to represent various different classes of concurrent programs (and their proofs) for this purpose.

Here, we will focus on one instance of this methodology [11] from our past work, and discuss the ideas behind algorithmic solutions to the steps above over a running example.

Consider the program in Figure 2 where the illustrated code is executed by unboundedly many threads. This program implements the ticket mutual exclusion protocol. The safety property of interest is that no two threads are ever simultaneously in their critical sections. This program has two global integer typed variables \( s \) and \( t \), while it has unboundedly many integer typed local variables \( m \), one per each thread. The challenge of proving this property for this program, beyond the standard challenges of dealing with shared memory and infinite-state programs, is that the local variables of unboundedly many threads effectively require an unbounded sized memory to keep track of their values in the proof.

Figure 2 also illustrates the control flow automaton for each thread in this program. Let us start by looking at a run of this program that would lead to two threads being in a
critical section at the same time. The run is a string accepted by the parallel composition of unboundedly many of versions of the control flow automaton illustrated in Figure 2. Below,
is a run of the program with only two participating threads (blue and red), where each
instruction includes the id of the thread executing it in form of: \( i (i \in \{1, 2\}) \), and the two
local variables for the two threads are distinguished by giving them two different subscripts
for clarity:

\[
m_1 = t++ : 1; \quad m_2 = t++ : 2; \quad \text{assume}(m_1 \leq s) : 1; \quad \text{assume}(m_2 \leq s) : 2;
\]

Note that the run does not correspond to a real execution of the program; in other words,
it is infeasible. It is easy to argue that such a run is infeasible and therefore a spurious
counter example to violation of mutual exclusion. One possible proof for this is illustrated
in Figure 3. What can we learn from this proof that would help with the construction of a
proof of correctness for the entire program?

A natural and minimal unit of reasoning that can be extracted from a proof of the

correctness of a run is a set of Hoare triples. For example, the Hoare triples demonstrate in
Figure 3 can be learned from the proof of the run in the same figure. Consider a deductive
system in which these triples can be considered as axioms. What should be the inference
rules of this deductive system that would produce new judgements based on a
finite set of base axioms? The answer lies in the definition of a proof space [11].

A proof space \( \mathcal{H} \) is a set of valid Hoare triples which is:

- Closed under symmetry: Let \( \pi : \mathbb{N} \to \mathbb{N} \) be any index permutation.

\[
\{ \phi \} \langle \sigma_1 : 1 \rangle \cdots \langle \sigma_n : 1 \rangle \langle \psi \rangle \in \mathcal{H} \implies \{ \phi[\pi] \} \langle \sigma_1 : \pi(1) \rangle \cdots \langle \sigma_n : \pi(n) \rangle \langle \psi[\pi] \rangle \in \mathcal{H}
\]

For example

\[
\begin{align*}
\{ s \leq m_1 \land m_1 < m_2 \} & \quad \text{assume}(m_2 \leq s); \quad \{ \text{false} \} \\
\{ s \leq m_3 \land m_3 < m_1 \} & \quad \text{assume}(m_3 \leq s); \quad \{ \text{false} \}
\end{align*}
\]

- Closed under conjunction:

\[
\text{If } \{ \phi \} \quad \{ \psi \} \in \mathcal{H} \land \{ \phi' \} \quad \{ \psi' \} \in \mathcal{H} \implies \{ \phi \land \phi' \} \quad \{ \psi \land \psi' \} \in \mathcal{H}
\]
For example

\[
\begin{align*}
\{ m_1 < t \} \\
m_3 &= t++; \\
\{ m_1 < m_3 \}
\end{align*}
\]

\[
\begin{align*}
\{ m_2 < t \} \\
m_3 &= t++; \\
\{ m_2 < m_3 \}
\end{align*}
\]

= Closed under sequencing:

\[
\begin{align*}
\{ \phi_0 \} \tau_0 \{ \phi_1 \} \in \mathcal{H} \land \{ \phi'_1 \} \in \mathcal{H} \land \phi_1 \vdash \phi'_1 \implies \{ \phi_0 \} \tau_0 \tau_1 \{ \phi_2 \} \in \mathcal{H}
\end{align*}
\]

For example

\[
\begin{align*}
\{ s \leq t \} \\
m_1 &= t++; \\
\{ m_1 < t \}
\end{align*}
\]

\[
\begin{align*}
\{ m_1 < t \} \\
m_2 &= t++; \\
\{ m_2 < m_2 \}
\end{align*}
\]

\[
\begin{align*}
\{ s \leq t \} \\
m_1 &= t++; \\
\{ m_1 < t \}
\end{align*}
\]

\[
\begin{align*}
\{ m_2 = t++; \\
\{ m_1 < m_2 \}
\end{align*}
\]

Note how proofs of runs of arbitrary length can be constructed using the sequencing rule.

One can effectively consider symmetry, conjunction, and sequencing as the inference rules of the aforementioned deductive system, and a proof space as a theory of this system. Note that the symmetry rule is the one that facilitates reasoning about unboundedly many threads. Without it, sequencing and conjunction can be used to produce proofs of concurrent programs with a fixed number of threads [9].

The next natural question to ask is whether the closure (under symmetry, conjunction, and sequencing) of the set of Hoare triples in Figure 3 includes the proof of every run of the program in Figure 2. The answer is no. The complete proof corresponds to the basis illustrated in Figure 4 which includes one additional Hoare triple. This basis is a finite set of valid Hoare triples that generates the complete proof space for the ticket program through its closure under sequencing, symmetry, and conjunction rules. This additional Hoare triple can be mined out of a second sample run. Therefore, the proof of the program in Figure 2 is decomposed into two correctness arguments of two sample runs (one illustrated in Figure 3 and discussed above, and the other skipped for brevity).

It is important to note that the proof space does not make use of any of the exotic features that are common to logics for reasoning about concurrency, most notably auxiliary variables and universal quantification over threads.

How is the proof in Figure 4 checked? To reduce the problem of checking whether a candidate proof space serves as a correctness proof for a program, we developed the notion of predicate automata (PA), an infinite-state, infinite-alphabet generalization of alternating finite automata. Predicate automata are equipped with a finite vocabulary of predicates parameterized over natural numbers, and their states are propositions over this vocabulary. The transition function of a PA maps each predicate symbol and letter to a positive Boolean formula over its vocabulary. For example, the transition

\[
\delta(p(i, j), a : k) = (p(i, j) \land i \neq k) \lor (q(i) \land q(j) \land i = k)
\]
indicates that, if the PA is at state \( p(1,2) \) and reads \( a:2 \), then it transitions to \( p(1,2) \); if it then reads \( a:1 \), then it transitions to both the state \( q(1) \) and \( q(2) \). A finite basis \( B \) of Hoare triples gives rise to a predicate automaton which recognizes the same set of runs as the proof space generated by \( B \).

The proof checking problem for proof spaces reduces to the inclusion problem for predicate automata, which in turn reduces to the emptiness problem of predicate automata. Although this problem is undecidable in general, we proposed an algorithm which is a decision procedure for the special case of PAs where each predicate symbol in its vocabulary has arity at most one.

## 2 Dynamic memory

Separation logic is an extension of Hoare logic for reasoning about memory [21, 26]. It is based on the addition of a new logical connective, the separating conjunction \( P \ast Q \), which asserts that the heap can be split into two disjoint parts such that \( P \) holds in one and \( Q \) in the other. Separating conjunction allows for local reasoning in the sense that a specification of a program fragment need only involve the portion of the heap that is relevant to that fragment – the rest (the frame) is automatically proved to remain untouched via the Frame rule, stated below:

\[
\text{Frame} \quad \{P\} S \{Q\} \quad \{P \ast F\} S \{Q \ast F\}
\]

The capacity of separation logic for reasoning about disjointness of memory makes it appealing for reasoning about concurrency. This power can be illustrated by the appealing inference rule for parallel composition [19]:

\[
\text{Parallel Composition} \quad \{P_1\} S_1 \{Q_1\} \quad \{P_2\} S_2 \{Q_2\} \quad \{P_1 \ast P_2\} S_1 \parallel S_2 \{Q_1 \ast Q_2\}
\]

This rule gives concurrent separation logic (CSL) an intuitive way to prove properties of concurrent programs where threads act on disjoint memory. Generally, it is too much to expect that threads do not share memory, but a kind of disjointness can be achieved through the notion of ownership. The essential idea can be summarized as follows [20]:

- **Ownership Hypothesis.** A code fragment can access only those portions of state that it owns.
- **Separation Property.** At any time, the state can be partitioned into that owned by each process and each grouping of mutual exclusion.
type node = { data: int; next: node }
global top :: node

push(item):
  mtop := new node
  mtop.data := item
do:
  mtop.next := *top
while(!cas(top, mtop.next, mtop))

pop():
do:
  mtop := *top
if(mtop = null): break
mnext := mtop.next
while(!cas(top, mtop, mnext))
while(!cas(top, mtop.next, mtop))

Figure 5 A nonblocking stack.

The earliest variation of CSL managed ownership using resource invariants, following Owicki and Gries [22]. Subsequently, there has been a great deal of work on increasing the power of CSL, a recent survey of which appears in [3]. Each advance increases the scope of what is possible to prove. However, increasing the expressivity of these logics takes us farther away from the class of proofs that we know how to completely automate. We thus raise a challenge problem: can we automate proofs for concurrent heap-manipulating programs using the ideas of proof spaces and separation logic?

A natural starting point in adapting proof spaces to separation logic is to replace the Conjunction rule with the Frame rule. Call a set of valid separation logic triples closed under Sequencing, Symmetry, and Frame a separation proof space.

The power of separation proof spaces can be illustrated with the push routine of Treiber’s non-blocking stack [29], illustrated in Figure 5. The difficulty in verifying memory safety of push is that no process owns the global top variable. There are various methods that can be used to verify memory safety and even full functional correctness of the lock-free stack [31, 30, 27, 8]. However, such proofs are difficult to automate. Figure 6 pictures a basis for a proof space that proves the memory safety of push. We observe the following features:

1. Separating conjunction and the Frame rule allow the proof to scale to an arbitrary number of threads by maintaining disjointness between each thread’s local mtop.
2. Separation logic triples are ordinary – they do not make use of exotic features that require ingenuity (except perhaps the separating implication ¬∗, the adjoint of ∗).
3. The stack is not owned by any process – the fact that top points to the top of the stack is preserved by every statement of the program, either explicitly or by application of the Frame rule.

There are three challenges in making the combination of proof spaces and separation logic practical.

1. **Inference rules.** While it may be possible to get some mileage out of the Frame rule combined with Sequencing and Symmetry, we believe that additional inference rules will be required in order to verify systems of practical interest. For example, existentially quantified variables commonly used in separation logic, and some means for reasoning about them is likely required (in the Treiber stack, for instance, we resorted to using separating implication to avoid proliferation of existential variables).

2. **Proof generation.** While there has been a great deal of work on generating correctness proofs for single runs in Hoare logic, there is relatively little in separation logic [1]. The additional challenge imposed by proof spaces is that we need methods to synthesize proofs that decompose into small, re-usable components.

3. **Proof checking.** The problem of checking that a proof space can prove every run of a program correct is reduced to the emptiness problem of predicate automata. This
Figure 6 A basis for a separation proof space for Figure 5.

reduction exploits the duality between disjunction and conjunction to complement the
language of a predicate automaton recognizing all runs that are proved correct by a proof
space. This duality does not exist between disjunction and separating conjunction. Some
new mechanism (e.g., a new class of automaton) is likely needed to encode the proof
checking problem for separation proof spaces.

3 Recursion

The difficulty of analyzing concurrent programs with recursive procedures is well-known:
reachability is undecidable even for simple program models [25]. From the perspective
of constructing a proof by considering program runs, the set of runs of a (sequential)
program without procedures is regular, and with procedures, it is context-free. However
for concurrent programs, adding procedures jumps over the context-freedom, requiring a
multi-stack automaton to recognize the language of runs.

There are (at least) two possibilities for solving this problem: (1) use an automaton model
that is capable of recognizing the languages of multi-threaded recursive program runs; and
(2) use summaries to treat complete procedure calls as atomic actions, restoring regularity to
the program model. We will discuss both possibilities in turn.

The advantage of using an automaton model that can recognize the interleaved runs of
recursive procedures is that the proof system used for proving the correctness of those runs
may remain unchanged. In our framework the structure of the proof is divorced from the
structure of the program. For example, a proof space may very well cover all the runs of a
multi-threaded program with recursive procedures – the added difficulty is only in the proof
checking problem, where we must prove that it does. This problem requires us to devise
a class of automata capable of recognizing the language of runs in the program and the
language of runs proved correct in a proof.
In fact, predicate automata are already a suitable model for this task. Procedure calls can be simulated by fork/join parallelism: whenever a thread would make a recursive call, it instead forks a new thread to make the call on its behalf and then immediately joins to retrieve the result. Fork/join parallelism can be modelled with predicate automata using a binary predicate \textit{child}(i,j) that retains the information that thread \textit{j} was forked by thread \textit{i}. However, admitting a binary predicate into the vocabulary of a predicate automaton makes the emptiness problem undecidable. Thus, the research problem is: can we develop adequate semi-algorithms for emptiness checking for general predicate automata?

The classical tool for context-sensitive analysis of recursive procedures in sequential programs is to compute summaries that abstract the behaviour of each procedure call [28]. Shared memory concurrency breaks the abstraction barrier: the assumption that procedure calls execute atomically will cause us to miss potential bugs. In [24], for the case where concurrent programs have a fixed number of threads, a summarization-based technique is proposed where atomicity is argued through Lipton’s theory of reduction.

Although naive summarization (that is summarization that is unaware of the environment of concurrently executing threads) is unsound in general, in many cases a software developer’s mental model likely assumes some atomic specification for each procedure. Consider the example in Figure 7. This example was taken from [24] where the program consisted of parallel composition of two threads each executing the \textit{main} procedure, however the same assertion holds true for unboundedly many threads running \textit{main} in parallel. The main procedure has a local variable \textit{c} which is initialized nondeterministically. Procedure \textit{main} then calls \textit{foo} with \textit{c} as the actual parameter. Procedure \textit{foo} falls into infinite recursion if the parameter \textit{r} is 0. Otherwise, it increments global variable \textit{g} and returns. After returning from \textit{foo}, the main procedure asserts that \((g \geq 1)\). The specification that \textit{foo} increases \textit{g} by 1 or does not return at all is too strong; two threads may concurrently execute \textit{foo} and increment \textit{g} by 2. However, the specification \(\{g \geq 0\} \textit{foo}(c) \{g \geq 1\}\) is unproblematic (i.e. holds under environment interference) and strong enough to prove that the assertion in \textit{main} holds. Note that this is independent of an argument about the atomicity of the procedure, that is in the style of [24]. This is about atomicity of the specification in a concurrent environment independent of atomicity of the code. The relevant research question is then: how can our framework discover these atomic specifications for procedures and prove that they hold?
## References


Bidirectional Nested Weighted Automata

Krishnendu Chatterjee¹, Thomas A. Henzinger², and Jan Otop³

1 IST Austria, Klosterneuburg, Austria
krish.chat@ist.ac.at

2 IST Austria, Klosterneuburg, Austria
tah@ist.ac.at

3 University of Wrocław, Wrocław, Poland
jotop@cs.uni.wroc.pl

Abstract

Nested weighted automata (NWA) present a robust and convenient automata-theoretic formalism for quantitative specifications. Previous works have considered NWA that processed input words only in the forward direction. It is natural to allow the automata to process input words backwards as well, for example, to measure the maximal or average time between a response and the preceding request. We therefore introduce and study bidirectional NWA that can process input words in both directions. First, we show that bidirectional NWA can express interesting quantitative properties that are not expressible by forward-only NWA. Second, for the fundamental decision problems of emptiness and universality, we establish decidability and complexity results for the new framework which match the best-known results for the special case of forward-only NWA. Thus, for NWA, the increased expressiveness of bidirectionality is achieved at no additional computational complexity. This is in stark contrast to the unweighted case, where bidirectional finite automata are no more expressive but exponentially more succinct than their forward-only counterparts.

1998 ACM Subject Classification F.1.1 Models of Computation

Keywords and phrases weighted automata, nested weighted automata, complexity, bidirectional

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2017.5

1 Introduction

We study an extension of nested weighted automata (NWA) [13] that can process words in both directions. We show that this new and natural framework can express many interesting quantitative properties that the previous formalism could not. We establish decidability and complexity results of the basic decision problems for the new framework. We start with the motivation for quantitative properties, then describe NWA and our new framework, and finally the contributions.

Weighted automata. Automata-theoretic formalisms provide a natural way to express quantitative properties of systems. Weighted automata extend finite automata where every transition is assigned an integer number called weight. Thus a run of an automaton gives rise to a sequence of weights. A value function aggregates the sequence of weights into a single

* This research was supported in part by the Austrian Science Fund (FWF) under grants S11402-N23,S11407-N23 (RISE/SHiNE) and Z211-N23 (Wittgenstein Award), ERC Start grant (279307: Graph Games), Vienna Science and Technology Fund (WWTF) through project ICT15-003 and by the National Science Centre (NCN), Poland under grant 2014/15/D/ST6/04543.

© Krishnendu Chatterjee, Thomas A. Henzinger, and Jan Otop;
licensed under Creative Commons License CC-BY

28th International Conference on Concurrency Theory (CONCUR 2017).
Editors: Roland Meyer and Uwe Nestmann; Article No. 5; pp. 5:1–5:16

Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany
value. For non-deterministic weighted automata, the value of a word $w$ is the infimum value of all runs over $w$. First, weighted automata were studied over finite words with weights from a semiring, and ring multiplication as value function [19], and later extended to infinite words with limit averaging or supremum as value function [12, 11, 10]. While weighted automata over semirings can express several quantitative properties [22], they cannot express long-run average properties that weighted automata with limit averaging can [12]. However, even weighted automata with limit averaging cannot express some basic quantitative properties (see [13]).

Nested weighted automata. A natural extension of weighted automata is to add nesting, which leads to nested weighted automata (NWA) [13]. A nested weighted automaton consists of a master automaton and a set of slave automata. The master automaton runs over input infinite words. At every transition the master can invoke a slave automaton that runs over a finite subword of the infinite word, starting at the position where the slave automaton is invoked. Each slave automaton terminates after a finite number of steps and returns a value to the master automaton. Each slave automaton is equipped with a value function for finite words, and the master automaton aggregates the returned values from slave automata using a value function for infinite words.

Advantages of NWA. We discuss the various advantages of NWA.

1. For Boolean finite automata, nested automata are equivalent to the non-nested counterpart, whereas NWA are strictly more expressive than non-nested weighted automata [13, Example 5]. NWA provide a specification framework where many basic quantitative properties can be expressed, which cannot be expressed by weighted automata [13].

2. NWA provide a natural and convenient way to express quantitative properties. Every slave automaton computes a subproperty, which is then combined using the master automaton. Thus NWA allow to decompose properties conveniently, and provide a natural framework to study quantitative run-time verification.

3. Finally, subclasses of NWA are equivalent in expressive power with automata with monitor counters [16], and thus they provide a robust framework to express quantitative properties.

Bidirectional NWA. Previous works considered slave automata that can only process input words in the forward direction (forward-only NWA). However, to specify quantitative properties, it is natural to allow slave automata to run backwards, for example, to measure the maximal or average time between a response and the preceding request. In this work we consider this natural extension of NWA, namely bidirectional NWA, where slave automata can process words in the forward as well as the backward direction.

Natural properties. First, we show that many natural properties can be expressed in the bidirectional NWA framework. We present two examples below (details in Section 3).

1. Average energy level. Consider a quantitative setting where each weight represents energy gain or consumption, and thus the sum of weights represents the energy level. To express the average energy level property, the master automaton has long-run average as the value function, and at every transition it invokes a slave automaton that walks backward with sum value function for the weights. Thus the average energy level property is naturally expressed by NWA with backward-walking slave automata, while this property is not expressible by NWA with forward-walking slave automata.
2. **Data-consistency property (DCP).** Consider the data-consistency property (DCP) where the input letters correspond to reads, writes, null instructions, and commits. For each read, the distance to the previous commit measures how fresh is the read with respect to the last commit, and this can be measured with a backward-walking slave automaton. For each write, the distance to the next commit measures how fresh is the write with respect to the following commit, and this can be measured with a forward-walking slave automaton. Thus the average freshness, called DCP, is expressed with bidirectional NWA. Moreover, the DCP can neither be expressed by NWA with only forward-walking slave automata nor by NWA with only backward-walking slave automata.

**Our contributions.** We propose bidirectional NWA as a specification framework for quantitative properties. First, we show that the classes of forward-only NWA and backward-only NWA have incomparable expressiveness, and bidirectional NWA strictly generalize both classes. Second, we establish complexity of the emptiness and universality problems for bidirectional NWA, where we consider the limit-average value function for the master automaton and for the slave automata we consider standard value functions for finite words (such as min, max, and variants of sum). The obtained complexity results coincide with the results for forward-only NWA, and range from $\text{NLogSpace}$-complete, $\text{PTime}$ to $\text{PSpace}$-complete to $\text{ExpSpace}$. However the proofs for bidirectional NWA are much more involved than forward-only NWA. Thus bidirectional NWA have all the advantages of NWA but provide a more expressive framework for natural quantitative properties. Moreover, the added expressiveness of bidirectionality is achieved with no increase in the computational complexity of the decision problems (Table 1). We highlight two significant differences as compared to the unweighted case: (1) In the unweighted case bidirectionality does not change expressiveness, whereas we show for NWA it does; and (2) in the unweighted case for deterministic automata bidirectionality leads to exponential succinctness and increase in complexity of the decision problems, whereas for NWA bidirectionality does not change the computational complexity. Thus the combination of nesting and bidirectionality is very interesting in the weighted automata setting, which we study in this work.

**Related works.** Quantitative automata and logic have been extensively studied in recent years in many different contexts [19, 12, 4, 2]. The book [19] presents an excellent collection of results of weighted automata on finite words. Weighted automata on infinite words have been studied in [12, 11, 20]. Weighted automata over finite words extended with monitor counters have been considered (under the name of cost register automata) in [3, 21]. A version of nested weighted automata over infinite words has been studied in [6], and nested weighted automata over infinite words has been studied in [13, 15, 14]. Several quantitative logics have also been studied, such as [5, 7, 1]. However, none of these works consider the rich and expressive formalism of quantitative properties expressible by NWA with slaves that walk both forward and backward, retaining decidability of the basic decision problems.

In the main paper, we present the key ideas and main intuitions of the proofs of selected results, and detailed proofs are presented in the full version [17]. Moreover, to ease of presentation we focus on bidirectional NWA where each slave automaton is either forward walking or backward walking. We can allow slave automata that change direction while running, i.e., allow two-way slave automata and obtain the same complexity results; we discuss nested weighted automata with two-way slave automata in the full version [17].
2 Definitions

2.1 Words and automata

Words. We consider a finite alphabet of letters \( \Sigma \). A word over \( \Sigma \) is a (finite or infinite) sequence of letters from \( \Sigma \). We denote the \( i \)-th letter of a word \( w \) by \( w[i] \), and for \( i < j \) we define \( w[i..j] \) as the word \( w[i]w[i+1]...w[j] \). The length of a finite word \( w \) is denoted by \( |w| \) and the length of an infinite word \( w \) is \( |w| = \infty \). For an infinite word \( w \), word \( w[i..\infty] \) is the suffix of \( w \) with first \( i - 1 \) letters removed. For a finite word \( w \) of length \( k \), we define the reverse of \( w \), denoted by \( w^R \), as the word \( w[k..1] = w[1..k] \).

Labeled automata. For a set \( X \), an \( X \)-labeled automaton \( A \) is a tuple \( \langle \Sigma, Q, Q_0, \delta, F, C \rangle \), where (1) \( \Sigma \) is the alphabet, (2) \( Q \) is a finite set of states, (3) \( Q_0 \subseteq Q \) is the set of initial states, (4) \( \delta \subseteq Q \times \Sigma \times Q \) is a transition relation, (5) \( F \) is a set of accepting states, and (6) \( C : \delta \rightarrow X \) is a labeling function. A labeled automaton \( \langle \Sigma, Q, \{q_0\}, \delta, F, C \rangle \) is deterministic if and only if \( \delta \) is a function from \( Q \times \Sigma \) into \( Q \) and \( Q_0 \) is a singleton.

Semantics of (labeled) automata. A run \( \pi \) of a (labeled) automaton \( A \) on a word \( w \) is a sequence of states of \( A \) of length \( |w| + 1 \) such that \( \pi[0] \) belongs to the initial states of \( A \) and for every \( 0 \leq i \leq |w| - 1 \) we have \( \langle \pi[i], w[i+1], \pi[i+1] \rangle \) is a transition of \( A \). A run \( \pi \) on a finite word \( w \) is accepting if and only if the last state \( \pi[|w]| \) of the run is an accepting state of \( A \). A run \( \pi \) on an infinite word \( w \) is accepting if and only if some accepting state of \( A \) occurs infinitely often in \( \pi \). For an automaton \( A \) and a word \( w \), we define \( \text{Acc}(w) \) as the set of accepting runs on \( w \). Note that for deterministic automata, every word \( w \) has at most one accepting run \((|\text{Acc}(w)| \leq 1)\).

Weighted automata and their semantics. A weighted automaton is a \( \mathbb{Z} \)-labeled automaton, where \( \mathbb{Z} \) is the set of integers. The labels are called weights. We define the semantics of weighted automata in two steps. First, we define the value of a run. Second, we define the value of a word based on the values of its runs. To define values of runs, we will consider value functions \( f \) that assign real numbers to sequences of integers. Given a non-empty word \( w \), every run \( \pi \) of \( A \) on \( w \) defines a sequence of weights of successive transitions of \( A \), i.e., \( C(\pi) = (C(\pi[i-1], w[i], \pi[i]))_{0 \leq i \leq |w|} \); and the value \( f(\pi) \) of the run \( \pi \) is defined as \( f(C(\pi)) \). We denote by \( C(\pi)[i] \) the weight of the \( i \)-th transition, i.e., \( C(\pi[i-1], w[i], \pi[i]) \). The value of a non-empty word \( w \) assigned by the automaton \( A \), denoted by \( L_A(w) \), is the infimum of the set of values of all accepting runs; i.e., \( \inf_{\pi \in \text{Acc}(w)} f(\pi) \), and we have the usual semantics that the infimum of the empty set is infinite, i.e., the value of a word that has no accepting run is infinite. Every run \( \pi \) on the empty word has length 1 and the sequence \( C(\pi) \) is empty, hence we define the value \( f(\pi) \) as an external (not a real number) value \( \bot \). Thus, the value of the empty word is either \( \bot \), if the empty word is accepted by \( A \), or \( \infty \) otherwise. To indicate a particular value function \( f \) that defines the semantics, we call a weighted automaton \( A \) with value function \( f \) an \( f \)-automaton.

Value functions. For finite runs we consider the following classical value functions: for runs of length \( n + 1 \) we have

- **Max and min**: \( \max_{n+1}(C(\pi))[i] = \max_{i=1}^{n+1}(C(\pi))[i] \) and \( \min_{n+1}(C(\pi))[i] = \min_{i=1}^{n+1}(C(\pi))[i] \).
- **Sum and absolute sum**: the sum function \( \text{SUM}(\pi) = \sum_{i=1}^{n}(C(\pi))[i] \), the absolute sum \( \text{SUM}^+(\pi) = \sum_{i=1}^{n} |C(\pi)[i]| \), where \( |x| \) is the absolute value of \( x \).
Variants of bounded sum: we consider a family of functions called the (variants of) bounded sum value function \( \text{Sum}_L^U \). Each of these functions returns the sum if all the partial sums are in the interval \([L, U]\), otherwise there are many possibilities which lead to multiple variants. For example, we can require that for all prefixes \( \pi' \) of \( \pi \) we have \( \text{Sum}(\pi') \in [L, U] \). We can impose a similar restriction on all suffixes, all infixes etc. Moreover, if partial sums are not contained in \([L, U]\), a bounded sum can return \( \infty \), the first violated bound, etc.

For infinite runs we consider:

- Limit average: \( \text{LimAvg}(\pi) = \liminf_{k \to \infty} \frac{1}{k} \cdot \sum_{i=1}^{k} (C(\pi))[i] \).

### 2.2 Nested weighted automata

Nested weighted automata (NWA) have been introduced in [13] and originally allowed slave automata to move only forward. The variant we define here allow two types of slave automata, forward walking and backward walking. The original definition of NWA from [13] is versatile and hence it can be seamlessly extended to the case with bidirectional (forward-and backward-walking) slave automata. We follow the description of [13].

**Informal description.** A nested weighted automaton consists of a labeled automaton over infinite words, called the master automaton, a value function \( f \) for infinite words, and a set of weighted automata over finite words, called slave automata. A nested weighted automaton can be viewed as follows: given a word, we consider the run of the master automaton on the word, but the weight of each transition is determined by dynamically running slave automata; and then the value of a run is obtained using the value function \( f \). That is, the master automaton proceeds on an input word as an usual automaton, except that before taking a transition, it starts a slave automaton corresponding to the label of the current transition. The slave automaton starts at the current position of the master automaton in the input word and works on some finite part of it. There are two types of slave automata: (a) forward walking, which move onward the input word (toward higher positions), and (b) backward walking, which move towards the beginning of the input word. Once a slave automaton finishes, it returns its value to the master automaton, which treats the returned value as the weight of the current transition that is being executed. The slave automaton might immediately accept and return value \( \bot \), which corresponds to a silent transition, i.e., transition with no weight. If one of slave automata rejects, the nested weighted automaton rejects. We present two examples of properties expressible by NWA. Additional examples are presented in Section 3.

**Example 1 (Average response time and its dual).** Consider infinite words over \( \{r, g, \#\} \), where \( r \) represents requests, \( g \) represents grants, and \( \# \) represents idle. A basic and interesting property is the average number of letters between a request and the corresponding grant, which represents the long-run average response time (ART) of the system. This property cannot be expressed by a non-nested automaton [13]. ART can be expressed by a deterministic nested weighted automaton, which basically implements the definition of ART. This automaton invokes at every request a forward-walking slave automaton with \( \text{Sum}^+ \) value function, which counts the number of events until the following grant. On the other events the NWA takes silent transitions. Finally, the master automaton applies \( \text{LimAvg} \) value function to the values returned by slave automata. Figure 1 presents a run of the NWA computing ART.

We define the average workload property (AW), which measures the average number of pending requests. The average is computed over all positions in a word. Intuitively, if we
Nested weighted automata. A nested weighted automaton (NWA) with bidirectional slave automata is a tuple \( \langle A_{\text{mas}}; f; B_{-m}, \ldots, B_{0}, \ldots, B_{l} \rangle \), with \( m, l \in \mathbb{N} \) where (1) \( A_{\text{mas}} \), called the master automaton, is a \([-m, \ldots, l]\)-labeled automaton over infinite words (the labels are the indexes of automata \( B_{-m}, \ldots, B_{l} \)), (2) \( f \) is a value function on infinite words, called the master value function, and (3) \( B_{-m}, \ldots, B_{l} \) are weighted automata over finite words called slave automata. Intuitively, an NWA can be regarded as an \( f \)-automaton whose weights are dynamically computed at every step by the corresponding slave automaton. The automata \( B_{-m}, \ldots, B_{-1} \) (resp., \( B_{l}, \ldots, B_{0} \)) are called backward walking (resp., forward walking) slave automata. We refer to NWA with both forward and backward walking slave automata as bidirectional NWA. The automaton \( B_{0} \) immediately accepts and returns no weight; it is used to implement silent transitions, which have no weight. We define an \((f; g)\)-automaton as an NWA where the master value function is \( f \) and all slave automata are \( g \)-automata.

Semantics: runs and values. A run of \( A \) on an infinite word \( w \) is an infinite sequence \((\Pi, \pi_1, \pi_2, \ldots)\) such that (1) \( \Pi \) is a run of \( A_{\text{mas}} \) on \( w \); (2) for every \( i > 0 \) the label \( j =
\( C(\Pi[i-1], w[i], \Pi[i]) \) pointers at a slave automaton and (a) if \( j < 0 \), then \( \pi_i \) is a run of the automaton \( B_j \) on some prefix of the reverse word \((w[1, i])^R\), and (b) if \( j \geq 0 \), then \( \pi_i \) is a run of the automaton \( B_j \) on some finite prefix of \( w[i, \infty] \). The run \((\Pi, \pi_1, \pi_2, \ldots)\) is accepting if all runs \( \pi_i, \pi_1, \pi_2, \ldots \) are accepting (i.e., \( \Pi \) satisfies its acceptance condition and each \( \pi_1, \pi_2, \ldots \) ends in an accepting state) and infinitely many runs of slave automata have length greater than 1 (the master automaton takes infinitely many non-silent transitions). The value of the run \((\Pi, \pi_1, \pi_2, \ldots)\) is defined as \( \text{sil}(f)(v(\pi_1)v(\pi_2)\ldots) \), where \( v(\pi_i) \) is the value of the run \( \pi_i \) in the corresponding slave automaton, and \( \text{sil}(f) \) is the value function that takes its input sequence, removes \( \perp \) symbols and applies \( f \) to the remaining sequence. The value of a word \( w \) assigned by the automaton \( A \), denoted by \( L_A(w) \), is the infimum of the set of values of all accepting runs. We require accepting runs to contain infinitely many non-silent transitions as \( f \) is a value function over infinite sequences, hence the sequence \( v(\pi_1)v(\pi_2)\ldots \) with \( \perp \) removed must be infinite.

**Deterministic nested weighted automata.** An NWA \( A \) is deterministic if (1) the master automaton and all slave automata are deterministic, and (2) in all slave automata, accepting states have no outgoing transitions. Intuitively, a slave automaton in an accepting state can choose (non-deterministically) to terminate or continue running; condition (2) removes this source of non-determinism.

**Width of NWA.** An NWA has width \( k \) if and only if in every run at every position at most \( k \) slave automata are active.

### 3 Examples

In this section we present several examples of quantitative properties that can be expressed with bidirectional NWA.

**Example 2 (Average energy level).** We consider the average energy level property studied in [18, 8]. Consider \( W \in \mathbb{N} \) and an alphabet \( \Sigma_W \) consisting of integers from interval \([-W, W]\). These letters correspond to the energy change, i.e., negative values represent energy consumption whereas positive values represent energy gain. For \( w \in \Sigma_W \) we define the energy level at \( i \) as the sum \( w[1] + \ldots + w[i] \). The average energy property (AE) is the limit average of the energy levels at every position. For example, the average energy level of \( 2(-1)3((11))2^2 \) is 4.

AE can be expressed by a \((\text{LimAvg}; \text{Sum})\)-automaton \( A \) with backward-walking slave automata, but it is not expressible by \((\text{LimAvg}; \text{Sum})\)-automata with forward-walking slave automata. To express AE, a \((\text{LimAvg}; \text{Sum})\)-automaton \( A \) with backward-walking slave automata invokes at every position a slave automaton, which runs backward to the beginning of the word and sums up all the letters. In contrast, \((\text{LimAvg}; \text{Sum})\)-automata with forward-walking slave automata can use finite memory of the master automaton, but finite prefixes influence only finitely many values returned by slave slave automata and the limit-average value function neglects finite prefixes. Formally, we can show with a simple pumping argument that for every \((\text{LimAvg}; \text{Sum})\)-automaton with forward-walking slave automata, among words \( w_i = 1^00^w \) there exists a pair of words with the same value. In contrast, all these words have different AE (AE of \( w_i \) is \( i \)).

AE property is often considered in conjunction with bounds on energy values. Typically, energy should not drop below some threshold, in particular, it should not be negative. In addition, the energy storage is limited, which motivates the upper bound on the stored energy,
where the excess energy is released. These two restrictions lead to the interval constraint on energy levels, i.e., we require the energy level at every position to belong to a given interval \([L, U]\), which results in a variant of the bounded sum \(\text{SUM}^{L, U}\).

▶ Example 3 (Data consistency). Consider a database server, which processes instructions grouped into transactions. There are four instructions: read \(r\), write \(w\), void \# and commit \(c\). The commit instruction applies all writes, finishes the current transaction and starts a new one. The read instructions refer to writes applied before the previous commit.

In the presence of multiple clients connected to the database, there are two options to achieve consistency. One option is to use locks that can limit concurrency. A second approach is optimistic concurrency which proceeds without locks, and then rolls back in case there was a collision between transactions. In order to limit the number of roll backs, it is preferred that the read instructions occur shortly after commit, while write instructions are followed by the commit instruction as quickly as possible. Formally, we define (a) consistency (or freshness) of a read instruction as the number of steps to the first preceding commit instruction, and (b) consistency of a write instruction as the number of steps to the following commit instruction. The data consistency property (DCP) of \(w\) is the limit average of consistency of reads and writes in \(w\).

DCP is expressed by the following deterministic (\(\text{LimAvg}; \text{SUM}^{+}\))-automaton \(A\) with bidirectional slave automata. On every read \(r\) (resp., \(w\)), the NWA \(A\) invokes a slave automaton which walks backward (resp., forward) and counts the number of steps to the first encountered \(c\). On the remaining instructions \(c, \#\), the NWA \(A\) invokes a dummy slave automaton which corresponds to a silent transition.

▶ Example 4. Consider the framework of Example 3. For every position with read \(r\) or write \(w\) we define a regret at position \(i\) as the minimal distance to the preceding or the following commit \(c\). Intuitively, the regret corresponds to the number of instructions by which we have to prepone or postpone the commit to include the instruction at the current position. We consider the minimal regret property (MR) on words over \(\{r, w, c, \#\}\) defined the limit average over positions with \(r\) and \(w\) of the regret at these positions. MR can be expressed by a non-deterministic (\(\text{LimAvg}; \text{SUM}^{+}\))-automaton with bidirectional slave automata, which basically implements the definition of MR (the non-deterministic guess is whether it is the preceding or the following commit). The NWA invokes at every \(r\) or \(w\) position one of the following two slave automata \(B_B\), \(B_F\). The automaton \(B_B\) counts the number of steps to the preceding grant, while \(B_F\) counts the number of steps to the following grant.

4 Decision questions

For NWA with bidirectional slave automata, we consider the quantitative counterparts of the fundamental problems of emptiness and universality. The (quantitative) emptiness and universality problems are defined in the same way for weighted automata and all variants of NWA; in the following definition \(A\) denotes either a weighted automaton or an NWA.

Emptiness and universality. Given an automaton \(A\) and a threshold \(\lambda\), the emptiness (resp. universality) problem asks whether there exists a word \(w\) with \(L_A(w) \leq \lambda\) (resp., for every word \(w\) we have \(L_A(w) \leq \lambda\)).

▶ Remark. The emptiness and universality problems have been studied for forward-only NWA in [13].
For NWA the value functions considered for the master automaton are the infimum (or limit-infimum), the supremum (or limit-supremum), and the limit-average. For all the decidability results for the infimum (limit-infimum) and the supremum (limit-supremum) value functions the techniques are similar to unweighted automata [13], which can be easily adapted to the bidirectional framework. Hence in the sequel we only focus on bidirectional NWA with the limit-average value function for the master automaton.

Moreover, we study only the emptiness problem for the following reasons. First, for the deterministic case the emptiness and the universality problems are similar and hence we focus on the emptiness problem. Second, in the non-deterministic case the universality problem is already undecidable for LimAvg-automata even with no nesting [9].

### 4.1 The minimum, maximum and bounded sum value functions

First, we show that for $g$ being Min, Max, or a variant of the bounded sum value function $\text{Sum}_{L,U}$, the emptiness problem for (LimAvg; $g$)-automata with bidirectional slave automata is decidable in PSPACE. To show that, we prove a stronger result, i.e., every (LimAvg; $g$)-automaton can be effectively transformed to a LimAvg-automaton of exponential size.

**Key ideas.** Weighted automata with value functions Min, Max, Sum$_{L,U}$ are close to (non-weighted) finite-state automata. In particular, these automata have finite range and for each value $\lambda$ from the range, the set of words of value $\lambda$ is regular. Thus, instead of invoking a slave automaton, the master automaton can non-deterministically pick value $\lambda$ and verify that the value returned by this slave automaton is $\lambda$. For backward-walking slave automata the guessing can be avoided as the master automaton can simulate (the reverse of) runs of all backward-walking slave automata until the current position. Thus, we can eliminate slave automata from NWA, i.e., we transform such NWA to weighted automata. Formally, we show that for $g \in \{\text{Min}, \text{Max}, \text{Sum}_{L,U}\}$, every (LimAvg; $g$)-automaton with bidirectional slave automata can be transformed into an equivalent LimAvg-automaton of exponential size.

The emptiness problem for non-deterministic LimAvg-automata is in NLOGSPACE (assuming weights given in unary) and hence we have the containment part in the following Theorem 5. The hardness part follows from PSPACE-hardness of the emptiness problem for (LimAvg; $g$)-automata with forward-walking slave automata only [13].

**Theorem 5.** Let $g \in \{\text{Min}, \text{Max}, \text{Sum}_{L,U}\}$. The emptiness problem for non-deterministic (LimAvg; $g$)-automata with bidirectional slave automata is PSPACE-complete.

**Note.** The complexity in Theorem 5 does not depend on encoding of weights in slave automata, i.e., the problem is PSPACE-hard even for a fixed set of weights, and it remains in PSPACE for weights encoded in binary.

The average energy property from Example 2 with bounds on energy levels can be expressed with (LimAvg; Sum$_{L,U}$)-automata. The emptiness problem for these automata is decidable by Theorem 5.

**Remark (Parametrized complexity).** If we assume that the size of slave automata in Theorem 5 is bounded by a constant, the complexity of the emptiness problem drops to NLOGSPACE-complete. NLOGSPACE-hardness follows from NLOGSPACE-hardness of the emptiness problem for LimAvg-automata, which can be considered as a special case of NWA.

The results of this section apply to general bidirectional NWA. In the following section we consider bidirectional NWA with the sum value function, where we consider additional
restrictions of finite width (Section 5) and bounded width (Section 6). We also justify in Remark 5.1 that the finite width restriction is natural.

5 Finite-width case

In this section we study NWA satisfying the finite width condition. First, we briefly discuss the finite-width condition and argue that it is a natural restriction. Next, we show that the emptiness problem for (finite-width) \((\text{LimAvg}; \text{Sum}^+)\)-automata with bidirectional slave automata is decidable in \(\text{ExpSpace}\). We conclude this section with the expressiveness results; we show that classical NWA with forward-walking slave automata and NWA with backward-walking slave automaton have incomparable expressive power. Hence, (finite-width) \((\text{LimAvg}; \text{Sum}^+)\)-automata with bidirectional slave automata are strictly more expressive than NWA with one-direction slave automata.

5.1 The finite-width condition

Finite width. An NWA \(A\) has finite width if and only if in every accepting run of \(A\) at every position at most finitely many slave automata are active. Classical NWA with forward-walking slave automata only have finite width. Indeed, in any run, at any position \(i\) at most \(i\) slave automata can be active.

Example 6. Consider an NWA over \(\{a,b\}\) such that the master automaton accepts a single word \(ab\omega\) and all slave automata are backward walking and accept words \(b^*a\). All slave automata terminate at the first position of \(ab\omega\) and hence this NWA does not have finite width.

The automata expressing properties from Examples 1, 3 and 4 are finite-width \((\text{LimAvg}; \text{Sum}^+)\)-automata with bidirectional slave automata. Observe that an NWA does not have finite width if and only if it has an accepting run, in which at some position \(i\) infinitely many backward-walking slave automata terminate.

Remark (Finite width is natural for positive sum). Let \(A\) be a \((\text{LimAvg}; \text{Sum}^+)\)-automaton with bidirectional slave automata. Except for degenerate cases, runs of \(A\), which do not have finite width, have infinite value. Indeed, consider a run \(\pi\) and a position \(i_0\) at which infinitely many automata are active. Since only finitely many forward-walking slave automata are active at \(i_0\), infinitely many of them are backward-walking and for some position \(i < i_0\), infinitely many slave automata \(S\) terminate at position \(i\). Then, one of the following holds: either that value of this run is infinite or one of the following two degenerate cases happen: (a) The slave automata from \(S\) are invoked with zero density (i.e., if consider the longrun-average of the frequency of invoking slave automata, then it is zero). This situation represents that monitoring with slave automata happens with vanishing frequency which is a degenerate case. (b) The values returned by the slave automata from \(S\) are bounded. It follows that these automata take transitions of non-zero weight only in some finite subword \([i,j]\) of the input word \(w\). This situation represents monitoring of an infinite sequence, in which all events past position \(j\) are irrelevant. This is a degenerate case in the infinite-word case.

The finite-width property does not depend on weights and hence we can construct an exponential-size Büchi automaton \(\mathcal{A}\), which simulates runs of a given NWA \(A\). Having \(\mathcal{A}\), we can check whether it has a run corresponding to an accepting run of \(A\), in which infinitely many backward-walking slave automata terminate at the same position. This check can be done in logarithmic space and hence checking the finite-width property is in \(\text{PSPACE}\).
A simple reduction from the non-emptiness problem for NWA shows \( \text{PSPACE} \)-hardness of checking the finite-width property.

**Theorem 7.** The problem whether a given NWA has finite width is \( \text{PSPACE} \)-complete.

### 5.2 The absolute sum value function

We present the main result on NWA of finite width.

**Theorem 8.** The emptiness problem for finite-width \((\text{LimAvg}; \text{Sum}^+)\)-automata with bidirectional slave automata is \( \text{PSPACE} \)-hard and it is decidable in \( \text{ExpSpace} \).

**Key ideas.** \( \text{PSPACE} \)-hardness follows from \( \text{PSPACE} \)-hardness of the emptiness problem for \((\text{LimAvg}; \text{Sum}^+)\)-automata with forward-walking slave automata only. Containment in \( \text{ExpSpace} \) is shown by reduction to the bounded-width case, which is shown decidable in the following section (Theorem 13). We briefly describe this reduction. Consider a finite-width \((\text{LimAvg}; \text{Sum}^+)\)-automaton \( A \) with bidirectional slave automata. First, we observe that without loss of generality, we can assume that \( A \) is deterministic. Second, we observe that in every word accepted by \( A \), at almost every position \( i \) there exists a barrier, which is a word \( u \) such that (a) the word \( w' = w[1, i]uw[i + 1, \infty] \), i.e., \( w \) with \( u \) inserted at position \( i \), is accepted by \( A \), and the runs on \( w \) and \( w' \) coincide except for positions in \( w' \) corresponding to \( u \), (b) in the run on \( w' \), backward-walking slave automata active at the end of \( u \) terminate within \( u \), (c) in the run on \( w' \), forward-walking slave automata active at the beginning of \( u \) terminate within \( u \), and (d) \( u \) has exponential length. Basically, active slave automata cannot cross \( u \) in \( w' \) and in the effect insertion of \( u \) bounds the number of active slave automata. Existence of barriers follows from the finite-width property of \( A \).

We insert barriers in \( w \) to reduce the number of active slave automata. We show that if at position \( i \) in \( w \), exponentially many active slave automata accumulate exponential weight past crossing \( i \) (some slave automata walk forward while other backwards), all partial averages (of values returned by slave automata) in \( w' \) are bounded by the corresponding partial averages in \( w \). We conclude that for every word \( w \), there exists a word \( w' \) such that (i) at every position at most exponentially many slave automata accumulate at least exponential values, and (ii) the value of \( w' \) does not exceed the value of \( w \). Thus, to compute the infimum over all runs of \( A \), we can focus on runs in which at every position at most exponentially many slave automata accumulate at least exponential values. Runs of slave automata in which they accumulate bounded (exponential) values can be eliminated as in Theorem 5, i.e., we can construct an exponential size NWA \( A' \), which simulates \( A \), and such that its slave automata run as long as they can accumulate value exponential (in \( |A| \)) and otherwise they non-deterministically pick the remaining value and the master automaton verifies that the pick is correct. Therefore, the infimum over all runs of \( A \) coincides with the infimum over all runs of \( A' \) of width exponentially bounded.

**Remark (Parametrized complexity).** If we assume that the size of slave automata in Theorem 8 is bounded by a constant, the complexity of the emptiness problem drops to \( \text{NLogSpace} \)-complete. \( \text{NLogSpace} \)-hardness follows from \( \text{NLogSpace} \)-hardness of the emptiness problem for \( \text{LimAvg} \)-automata, which can be viewed as a special case of NWA.

### 5.3 Expressive power

DCP defined in Example 3 can be expressed by a deterministic finite-width \((\text{LimAvg}; \text{Sum}^+)\)-automaton with bidirectional slave automata. We show that both forward-walking and
backward-walking slave automata are required to express DCP. That is, we formally show that DCP cannot be expressed by any (non-deterministic) \( \text{LimAvg}; \text{Sum}^+ \)-automaton with slave automata walking in one direction only.

**Classes of NWA.** We define \( \mathcal{FB}(\text{LimAvg}; \text{Sum}^+) \) as the class of all finite-width (\( \text{LimAvg}; \text{Sum}^+ \))-automata with bidirectional slave automata. We define \( \mathcal{F}(\text{LimAvg}; \text{Sum}^+) \) (resp., \( \mathcal{B}(\text{LimAvg}; \text{Sum}^+) \)) as the subclass of \( \mathcal{FB}(\text{LimAvg}; \text{Sum}^+) \) consisting of NWA with forward-walking (resp., backward-walking) slave automata only.

We establish that classes \( \mathcal{F}(\text{LimAvg}; \text{Sum}^+) \) and \( \mathcal{B}(\text{LimAvg}; \text{Sum}^+) \) have incomparable expressive power and hence they are strictly less expressive than class \( \mathcal{FB}(\text{LimAvg}; \text{Sum}^+) \).

**Key ideas.** Consider a word \( w = (c\#^r_{2^K} c\#^{2^N_{2^K}})^\omega \) for some big \( K \) and much bigger \( N \). An NWA from \( \mathcal{B}(\text{LimAvg}; \text{Sum}^+) \) computes DCP of \( w \) by invoking (non-dummy) slave automata at every \( r \) letter and taking silent transitions on letters \( \#, c \). We show that an NWA \( A \) from \( \mathcal{F}(\text{LimAvg}; \text{Sum}^+) \) cannot invoke the right number of slave automata, even if it uses non-determinism. More precisely, we show that \( A \) computing DCP has to invoke at most \( O(K) \) non-dummy slave automata on average on subwords \( c\#^N_{2^K} c\#^{2^N_{2^K}} \). Since \( N \) is much bigger than \( K \), we conclude that \( A \) has a cycle over \( \# \) letters at which it takes only silent transitions and a cycle over \( r \) letters on which it increases the multiplicity of active slave automata. Using these two cycles, we construct a run of value smaller than \( \omega \) letters on which it increases the multiplicity of active slave automata. Therefore, an NWA from \( \mathcal{F}(\text{LimAvg}; \text{Sum}^+) \) cannot compute correctly DCP of words of the form \( w = (cw_{2^K} \#^N c\#^{2^K} \#^{2^N})^\omega \), while on these words DCP is expressible by an NWA from \( \mathcal{FB}(\text{LimAvg}; \text{Sum}^+) \).

\begin{itemize}
  \item **Lemma 9.** (1) DCP restricted to alphabet \( \{r, \#, c\} \) is expressed by an NWA from \( \mathcal{B}(\text{LimAvg}; \text{Sum}^+) \), but it is not expressible by NWA from \( \mathcal{F}(\text{LimAvg}; \text{Sum}^+) \). (2) DCP restricted to alphabet \( \{w, \#, c\} \) is expressed by an NWA from \( \mathcal{F}(\text{LimAvg}; \text{Sum}^+) \), but it is not expressible by NWA from \( \mathcal{B}(\text{LimAvg}; \text{Sum}^+) \).
\end{itemize}

The above lemma implies that DCP over alphabet \( \{r, w, \#, c\} \) is not expressible by any NWA from \( \mathcal{F}(\text{LimAvg}; \text{Sum}^+) \) nor from \( \mathcal{B}(\text{LimAvg}; \text{Sum}^+) \). In conclusion, we have:

\begin{itemize}
  \item **Theorem 10.** (1) \( \mathcal{FB}(\text{LimAvg}; \text{Sum}^+) \) and \( \mathcal{B}(\text{LimAvg}; \text{Sum}^+) \) have incomparable expressive power. (2) \( \mathcal{FB}(\text{LimAvg}; \text{Sum}^+) \) are strictly more expressive than \( \mathcal{F}(\text{LimAvg}; \text{Sum}^+) \) and \( \mathcal{B}(\text{LimAvg}; \text{Sum}^+) \).
\end{itemize}

### 6 Bounded-width case

In this section, we study \( \text{LimAvg}; \text{Sum} \)-automata with bidirectional slave automata, which have bounded width. The bounded width restriction has been introduced in [14] to improve the complexity of the emptiness problem and to establish decidability of the emptiness problem for \( \text{LimAvg}; \text{Sum} \)-automata. NWA considered in [14] have only forward-walking slave automata, while we extend these results to NWA with bidirectional slave automata. This extension preserves the complexity bounds from [14], i.e., the emptiness problem is in \( \text{PTime} \) for constant width and \( \text{PSPACE} \)-complete for width given in unary.

The bounded width restriction emerges naturally in examples presented so far. If we bound the number of pending requests, we can express ART and AW (Example 1) by automata of bounded width. If we bound the number of writes and reads between any two
We can show that the value of any dense run, is the graph of the master automaton. We compute the partial average over prefixes of some slave automata. Often in some accepting run, with the successor relation.

A \( k \)-configuration of a master automaton, and each \( q_1, \ldots, q_k \) is a state of \( A \) or \( \bot \). Given a run of \( A \), we say that \( (q; q_1, \ldots, q_k) \) is the \( k \)-configuration at position \( i \) in the run if \( q \) is the state of the master automaton at position \( i \) and there are \( l \leq k \) active slave automata at position \( i \), whose states are \( q_1, \ldots, q_l \) ordered by position of invocation (backward-walking slave automata are invoked past position \( i \)). If \( l < k \), then \( q_{l+1}, \ldots, q_k = \bot \). We say that a \( k \)-configuration \( C_2 \) is a successor of a \( k \)-configuration \( C_1 \) if there exists an accepting run of \( A \) and \( i > 0 \) such that \( C_1 \) is the \( k \)-configuration at \( i \) and \( C_2 \) is the \( k \)-configuration at \( i + 1 \).

The graph of \( k \)-configurations of \( A \) is the set of \( k \)-configurations of \( A \), which occur infinitely often in some accepting run, with the successor relation.

Main ideas. In Example 11, the words attaining the infimum contain long blocks of letter \( b \), at which the NWA \( A \) is (virtually) in the same state, i.e., it loops in this state. On letters \( b \), the sum of all weights collected by all active slave automata is \( 2 \), i.e., automata \( B_a \), \( B_c \) collect weight \( 1 \) and \( B_b \) collect \( 0 \). However, in computing limit infimum over partial averages, we pick positions just before letter \( c \) as they correspond to the local minima, i.e., we compute the partial average over prefixes \( uab \), and hence the weights collected by \( B_c \) do not contribute to this partial average. Then, the sum of all weights collected by slave automata \( B_a \), \( B_b \) over a letter \( b \) is \( 1 \), which is equal to the least value of the limit infimum of the partial averages. In the following, we extend this idea and present the solution for all bounded-width (LimAvg; Sum)-automata with bidirectional slave automata. We show that the infimum over all words of a given NWA is the least average value over all cycles. In the following, we define appropriate notions of cycles of NWA and their average with exclusion of some slave automata.
Characteristics of cycles. Let $C$ be a cycle in a graph of $k$-configurations of $A$. Let $F$ (resp., $B$) be the set of forward-walking (resp., backward-walking) slave automata, which are active throughout $C$, i.e., which are not invoked nor terminated within $C$. A focus $F_C$ (for $C$) is a downward closed subset of $F$, i.e., it contains all automata from $F$ invoked before some position. We define a focused gain $\text{Gain}(C,F_C)$ as the sum of weights which automata from $F_C$ accumulate over $C$. A restriction $R$ (for $C$) is an upward closed subset of $B$, i.e., it contains all automata from $B$ invoked past certain position. We define an average weight of $C$ excluding $R$, denoted by $\text{AvgE}(C,R)$, as the sum of weights of all transitions of slave automata within $C$, except of transitions of slave automata from $R$, divided by the number of slave automata invoked within $C$.

Intuitively, a focused gain refers to the value, which forward-walking slave automata invoked before some position $i$, accumulate over the part of run corresponding to $C$ (see Figure 2). If the focused gain $\text{Gain}(C,F_C)$ is negative, then by pumping $C$ we can arbitrarily decrease the partial average of the values of slave automata invoked before $i$. In consequence, we can construct a run of the value $-\infty$. Formally, we define condition (*), which implies that there exists a run of value $-\infty$, as follows: (*) there exists a cycle $C$ in the graph of $k$-configurations of $A$ and a focus $F_C$ such that $\text{Gain}(C,F_C) < 0$.

If the focused gain of every cycle is non-negative, we need to examine averages of cycles, while excluding some backward-walking slave automata. The average weight with restriction corresponds to the partial average of values aggregated over $C$ by all slave automata invoked before position $j$ (which can be past $C$). Backward-walking slave automata in the restriction correspond to automata invoked past $j$, and hence their values do not contribute to the partial average (up to $i$) (see Figure 2). In Example 11, we compute the average of slave automata over letters $b$, but we exclude the backward-walking slave automaton invoked at the following letter $c$. Observe that for any cycle $C$ and any restriction $R$, having a run containing $C$ occurring infinitely often, we can repeat each occurrence of cycle $C$ sufficiently many times so that the partial average of values of slave automata up to position corresponding to $j$ becomes arbitrarily close to the average $\text{AvgE}(C,R)$. The resulting run contains a subsequence of partial averages convergent to $\text{AvgE}(C,R)$ and hence its value does not exceed $\text{AvgE}(C,R)$.

We can now state our key technical lemma. This lemma is a direct extension of an intuition behind computing the infimum over values of all words of the NWA $A$ from Example 11.

Lemma 12. Let $A$ be a (LIMAVG; SUM)-automaton of bounded width with bidirectional slave automata. (1) If condition (*) holds, then $A$ has a run of value $-\infty$. (2) If (*) does not hold, then the infimum $\text{inf}_w A(w)$ equals the infimum $\text{inf}_{C \in \Lambda} A(C)$, where $\Lambda$ is the set of all cycles $C$ in the graph of $k$-configurations of $A$.

If the width of $A$ is constant, then the graph of $k$-configurations has polynomial size in $|A|$ and it can be constructed in polynomial time by employing reachability checks on the set
Table 1 The complexity of the emptiness problem for \((\text{LimAvg}; g)\)-automata. The columns describe respectively: the value function \(g\), restrictions imposed on the problem, the complexity in the case with bidirectional slave automata, and the complexity in the previously studied [13, 14] case with only forward-walking slave automata. Results presented in this paper are boldfaced.

<table>
<thead>
<tr>
<th>Value func. (g)</th>
<th>Restrictions</th>
<th>Complexity Bidirectional</th>
<th>Complexity Forward case</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIN, MAX, SUM(^L,U)</td>
<td>None</td>
<td>PSpace-complete (Thm 5)</td>
<td>PSpace-complete [13]</td>
</tr>
<tr>
<td>SUM(^+)</td>
<td>finite width</td>
<td>PSpace-hard ExpSpace (Thm 8)</td>
<td>PSpace-hard ExpSpace [13]</td>
</tr>
<tr>
<td>SUM(^+), SUM</td>
<td>constant width unary weights</td>
<td>NLogSpace-complete (Thm 13)</td>
<td>NLogSpace-complete [14]</td>
</tr>
<tr>
<td>SUM(^+), SUM</td>
<td>constant width binary weights</td>
<td>PTime (Thm 13)</td>
<td>PTime [14]</td>
</tr>
<tr>
<td>SUM(^+), SUM</td>
<td>width given in unary</td>
<td>PSpace-complete (Thm 13)</td>
<td>PSpace-complete [14]</td>
</tr>
</tbody>
</table>

of all \(k\)-configurations w.r.t. to relaxation of the successor relation. Therefore, for every focus \(Fc\) and every \(k\)-configuration \(c\) we can check in polynomial time whether there exists a cycle \(C\) such that \(C[1] = c\) and \(\text{Gain}(C, Fc) < 0\). Thus, condition (1) can be check in logarithmic space assuming that weights are given in unary. If weights are given in binary, condition (1) can be checked in polynomial time. Checking condition (2) has the same complexity as condition (1). If the width \(k\) is given in unary in input, the graph of \(k\)-configurations is exponential in \(|A|\) and conditions (1) and (2) can be checked in polynomial space. Weights in this case are logarithmic in the size of the graph and hence changing representation from binary to unary does not affect the (asymptotic) size of the graph.

Theorem 13. The emptiness problem for \((\text{LimAvg}; \text{SUM})\)-automaton of width \(k\) with bidirectional slave automata is (a) NLogSpace-complete for constant \(k\) and weights given in unary, (b) in PTime for constant \(k\) and weights given in binary, and (c) PSpace-complete for \(k\) given in unary.

7 Discussion and Conclusion

Discussion. We established decidability of the emptiness problem for classes of bidirectional NWA, which include all NWA presented in the examples. An NWA from Example 2 is covered by Theorem 5, while NWA from Examples 1, 3 and 4 are covered by Theorem 8. The lower bounds in our results follow from the lower bounds of the special case of forward-only NWA. The established complexity (Table 1) coincide with the forward-only case.

Concluding remarks. In this work we present bidirectional NWA as a specification formalism for quantitative properties. There are several interesting directions for future work. The study of bidirectional NWA with other value functions is an interesting direction. The second direction of future work is to consider other formalism (such as a logical framework) which has the same expressive power as bidirectional NWA.

References


\textbf{k}-Bounded Petri Net Synthesis from Modal Transition Systems* \\

Uli Schlachter\textsuperscript{1} and Harro Wimmel\textsuperscript{2} \\

1 Department of Computing Science, Carl von Ossietzky Universität, Oldenburg, Germany  
uli.schlachter@informatik.uni-oldenburg.de \\
2 Department of Computing Science, Carl von Ossietzky Universität, Oldenburg, Germany  
harro.wimmel@informatik.uni-oldenburg.de \\

\textbf{Abstract} \\
We present a goal-oriented algorithm that can synthesise \textit{k}-bounded Petri nets ($k \in \mathbb{N}^+$) from hyper modal transition systems (hMTS), an extension of labelled transition systems with optional and required behaviour. The algorithm builds a potential reachability graph of a Petri net from scratch, extending it stepwise with required behaviour from the given MTS and over-approximating the result to a new valid reachability graph. Termination occurs if either the MTS yields no additional requirements or the resulting net of the second step shows a conflict with the behaviour allowed by the MTS, making it non-sythesisable.

\textbf{1998 ACM Subject Classification} D.2.2 Design Tools and Techniques, F.4.1 Mathematical Logic \\

\textbf{Keywords and phrases} Modal transition system, bounded Petri net, synthesis \\

\textbf{1 Introduction} \\
Petri net synthesis, or more precisely, the problem of finding an unlabelled Petri net implementing a given labelled transition system, goes back to Ehrenfeucht and Rozenberg \cite{13} and was recently comprehensively presented by Badouel, Bernardinello and Darondeau \cite{5}. Petri net synthesis not only yields implementations which are correct by design, but also allows to extract concurrency and distributability information from a sequential specification \cite{6, 8, 21}.

Modal transition systems (MTS) are a well-known and useful method for specifying systems \cite{17, 1, 9, 16}. They are an extension of labelled transition systems (LTS) that can distinguish between required behaviour (must edges) and optional behaviour (may edges). Hyper MTS (called disjunctive MTS in \cite{18}) are a further extension, where must edges are defined as hyper edges, requiring at least one out of (possibly) several actions. An LTS implements a (hyper) MTS if it allows the required behaviour and disallows any not specified (optional or required) behaviour, i.e. there may be more than one LTS implementing the same MTS.

Since modal transition systems are extensions of labelled transition systems, it has been suggested to extend Petri net synthesis to cover modal transition systems \cite{12, 5}. The question here is if a given modal transition system can be realised by a Petri net, which means that the reachability graph of a Petri net can implement the modal transition system,

* This work is supported by the German Research Foundation (DFG) projects ARS and ASYST, reference numbers Be 1267/15-1 and Be 1267/16-1.
and if this is answered positively, to produce such a Petri net. Recent results show that this question is undecidable even when asking for pure Petri nets [14] or for bounded Petri nets [20].

We will easily see that for a given \( k \in \mathbb{N}^+ \) the synthesis of a \( k \)-bounded Petri net from some MTS (or some other specification) is decidable, since there is only a finite number of different reachability graphs available for \( k \)-bounded Petri nets with a fixed set of transitions. This is, to our knowledge, the first general positive decidability result for Petri net synthesis from modal specifications. Such a brute force approach is not advisable, though, due to the state space explosion with growing \( k \) and number of transitions.

Instead, we introduce a goal-oriented algorithm for synthesising \( k \)-bounded Petri nets from hyper MTS, where the value \( k \) is an input to the algorithm. Our algorithm iteratively applies two operations until a fixed point is reached. The first operation adds missing edges so that its input implements a given modal transition system. The second operation produces minimal Petri net solvable over-approximations of otherwise Petri net unsolvable labelled transition systems. The algorithm will construct all minimal Petri net realisations of a given modal transition system, where minimality is defined with respect to an LTS homomorphism preorder, similar to the simulation preorder. This preorder implies language inclusion, which means that the algorithm also calculates minimal realisations with respect to language inclusion.

The paper is organised as follows: In Section 2 we present the basic terminology regarding LTS and Petri nets, including LTS homomorphisms, the construction of \( k \)-bounded regions, and the synthesis of \( k \)-bounded Petri nets from such regions. In Section 3 we introduce (hyper) MTS and their relation to LTS. Section 4 contains our main algorithm including some examples. In Section 5 we prove termination and correctness of our algorithm. Section 6 contains some thought on the algorithmic complexity of the algorithm and how to implement it.

## 2 Petri Net Synthesis

**Definition 1.** A-labelled transition system (LTS) \( A \) is a structure \( A = (Q, \Sigma, \rightarrow, q_0) \) where \( Q \) is a set of states, \( \Sigma \) is a set of actions, \( q_0 \in Q \) is the initial state and \( \rightarrow \subseteq Q \times \Sigma \times Q \) is a set of (action-)labelled edges. A tuple \((q, a, q') \in \rightarrow\) is also written as \( q \xrightarrow{a} q' \) and \( q \xrightarrow{\alpha} \) expresses that some \( q' \) with \( q \xrightarrow{\alpha} q' \) exists. This notation is canonically extended to words \( w \in \Sigma^* \) by inductively defining \( q \xrightarrow{w} q \) for all \( q \in Q \) and \( q \xrightarrow{\omega} q' \iff \exists q'' : q \xrightarrow{w} q'' \wedge q'' \xrightarrow{\alpha} q' \). An LTS is deterministic if it satisfies \( \forall(q, a, q'), (q, a, q'') \in \rightarrow : q' = q'' \). It is called totally reachable if \( \forall q \in Q : \exists w \in \Sigma^* : q_0 \xrightarrow{w} q \). It is called finite if \( Q \) and \( \Sigma \) (and hence also \( \rightarrow \)) are finite. The language \( L(A) \) of an LTS \( A \) is \( L(A) := \{ w \in \Sigma^* | q_0 \xrightarrow{w} \} \).

Given two LTS \( A_i = (Q_i, \Sigma_i, \rightarrow_i, q_{0i}) \) with \( i \in \{1, 2\} \), an LTS homomorphism from \( A_1 \) to \( A_2 \) is a function \( f : Q_1 \rightarrow Q_2 \) with \( f(q_{01}) = q_{02} \) such that \( q \xrightarrow{\alpha} q' \) implies \( f(q) \xrightarrow{\alpha} f(q') \) for all \( q, q' \in Q_i \) and \( a \in \Sigma \). If such a homomorphism \( f \) exists we write \( A_1 \sqsubseteq A_2 \) (via \( f \)). \( \sqsubseteq \) is reflexive (with the identity homomorphism id) and transitive (as homomorphisms are closed under composition). A bijection \( f \) is an isomorphism if both, \( f \) and \( f^{-1} \), are homomorphisms, in which case we abstract from state names of the isomorphic LTS \( A_1 \) and \( A_2 \) and identify them, writing \( A_1 = A_2 \).

**Remark.** Unless given explicitly, the components of an LTS \( A \) will be named canonically as \( A = (Q_A, \Sigma_A, \rightarrow_A, q_{0A}) \). The same notation will be used for other structures. For example, the set of places of a Petri net \( N \), which will be defined later, is referred to as \( P_N \). For
An example of a Petri net and its reachability graph is shown in Figure 1. The following lemma can easily be verified:

**Lemma 2.** Let \( A \) and \( B \) be LTS so that \( A \sqsubseteq B \) via \( f \) and \( A \sqsubseteq B \) via \( f' \). If \( A \) is totally reachable and \( B \) is deterministic, then \( f = f' \).

**Proof.** We prove by induction on the length of words \( w \in L(A) \) that if \( q_0,A \xrightarrow{w} q \), then \( f(q) = f'(q) \). Since \( A \) is totally reachable, we reach all states of \( A \) in this way, showing that \( f(q) = f'(q) \) for all \( q \in Q_A \). The induction basis shows the conclusion for the initial state by applying the definition of \( \sqsubseteq \): \( f(q_0,A) = q_0,B = f'(q_0,A) \).

For the induction step, assume that \( q_0,A \xrightarrow{w} q \) with \( f(q) = f'(q) \) and consider any \( q \xrightarrow{a} q' \). Since \( A \sqsubseteq B \) via \( f \) and \( f' \), we have \( f(q) \xrightarrow{a} f(q') \) and \( f'(q) \xrightarrow{a} f'(q') \). However, because \( B \) is deterministic, there cannot be two different states that are reached from \( f(q) = f'(q) \) via label \( a \). We conclude \( f(q') = f'(q') \).

**Lemma 3.** Let \( A \) and \( B \) be totally reachable and deterministic LTS so that \( A \sqsubseteq B \) via \( f_A \) and \( B \sqsubseteq A \) via \( f_B \). Then \( A = B \).

**Proof.** Consider \( A \sqsubseteq A \) (via \( \id \)) and \( A \sqsubseteq B \sqsubseteq A \) via \( f_B \circ f_A \). Since Lemma 2 is applicable, we conclude \( f_B \circ f_A = \id \), making \( f_A \) injective and \( f_B \) surjective. With an analogous argument for \( f_A \circ f_B = \id \) we see that \( f_A \) and \( f_B \) are both bijective homomorphisms with \( f_A^{-1} = f_B \), i.e. they are isomorphisms. Abstracting from state names for the isomorphic \( A \) and \( B \), we may write \( A = B \).

**Definition 4.** A Petri net is a tuple \( N = (P,T,F,M_0) \) where \( P \) and \( T \) are finite and disjoint sets of places and transitions, respectively, and \( F: ((P \times T) \cup (T \times P)) \to N \) is a flow relation specifying arc weights. \( M_0 \) is the initial marking where a marking is a mapping \( P \to \mathbb{N} \). A transition \( t \in T \) is enabled in a marking \( M \) if \( \forall p \in P: F(p,t) \leq M(p) \). An enabled transition can fire leading to the marking \( M' \) defined by \( M'(p) = M(p) - F(p,t) + F(t,p) \).

We write \( M[t],M' \) and this syntax is inductively extended to label sequences \( \sigma \in T^* \). The set of reachable markings of a Petri net \( N \) is \( \mathcal{E}(N) = \{ M \mid \exists \sigma \in T^*: M_0[\sigma]M \} \). A Petri net is called \( k \)-bounded by \( k \in \mathbb{N} \) if \( \forall M \in \mathcal{E}(N) \), \( p \in P: M(p) \leq k \). There is a self-loop around \( p \in P \) and \( t \in T \) if \( F(p,t) > 0 \land F(t,p) > 0 \). A Petri net is called pure if it does not contain any self-loops. The reachability graph of \( N \) is the LTS \( RG(N) = (\mathcal{E}(N),T,\to,M_0) \) with \( \to = \{ (M,t,M') \in \mathcal{E}(N) \times T \times \mathcal{E}(N) \mid M[t],M' \} \). An LTS \( A \) is called Petri net solvable if a Petri net \( N \) exists with \( A = RG(N) \). In this situation, \( N \) solves \( A \).

An example of a Petri net and its reachability graph is shown in Figure 1. The following lemma can easily be verified:
Lemma 5. For a Petri net $N$, $RG(N)$ is totally reachable and deterministic. If $N$ is $k$-bounded (for some $k \in \mathbb{N}^+$), $RG(N)$ is finite.

The next lemma shows that there are only finitely many LTS solvable via $k$-bounded Petri nets. As an immediate consequence, if we can decide whether the reachability graph of a specific $k$-bounded Petri net “implements” some specification, we can also decide whether any $k$-bounded Petri net fulfills this, i.e. brute force decisions are possible.

Lemma 6. For any given $k \in \mathbb{N}^+$ and finite set of transitions $T$, the number of structurally different reachability graphs of $k$-bounded Petri nets with transitions from $T$ is finite.

Proof. A weight $F(p,t) > k$ or $F(t,p) > k$ in a $k$-bounded Petri net means that the transition $t$ can never fire (or the bound would be violated). With weights in $\{0, \ldots, k\}$, a place can only be connected in finitely many ways to all the transitions. Exact duplicates of places do not have an effect on the structure of the reachability graph.

A place $p$ of a Petri net has a number of tokens $M(p)$ in each reachable marking $M$. The next definition formalises this concept on the reachability graph as a so-called region [7, 10, 11].

Definition 7. Let $k \in \mathbb{N}^+$. A $k$-bounded region of a finite LTS $A$ is a function $r: Q_A \rightarrow \{0, 1, \ldots, k\}$ satisfying $\forall q \in \Sigma_A: \forall q \xrightarrow{a} q'$, $\forall q'' \xrightarrow{a} q": r(q') - r(q) = r(q'') - r(q'')$, which means that edges with the same label have the same gradient, where the gradient of $a \in \Sigma_A$ is $\Delta_a(a) = r(q') - r(q)$ for any edge $q \xrightarrow{a} q'$. The value $\mu_r(a) = \min\{r(q) \mid q \xrightarrow{a} \}$ is the minimum value $r(q)$ of a state $q$ with an outgoing edge with label $a$ (generally assuming that every $a \in \Sigma_A$ occurs as a label of some edge).

A region corresponds to a possible place $p$ of a Petri net with initial marking $M_0(p) = r(q_0)$ and flow relation $F(p,t) = \mu_r(t)$, $F(t,p) = \Delta_r(t) + \mu_r(t)$. This correspondence allows to define the Petri net $N(R)$ generated by a set $R$ of regions of an LTS $A$ via $N(R) = (R, \Sigma_A, F, M_0)$ with $M_0$ and $F$ as above.

This definition adds the maximum possible number of self-loops to the generated place $p$, because this restricts the behaviour of the Petri net the most. Thus, places with fewer self-loops cannot be generated in this setting, but also will not be needed.

Lemma 8. Let $A$ and $B$ be finite LTS so that $A \subseteq B$ via $f$. If $r$ is a region of $B$, then $r' = r \circ f$ is a region of $A$.

Proof. Let $q \xrightarrow{a} q'$ and $\tilde{q} \xrightarrow{a} \tilde{q'}$ be two edges of $A$. Then, $f(q) \xrightarrow{a} f(q')$ and $f(\tilde{q}) \xrightarrow{a} f(\tilde{q'})$ are edges in $B$ and by the uniqueness of the gradient we get $r'(q') - r'(q) = r(f(q')) - r(f(q)) = r(f(q')) - r(f(\tilde{q})) = r'(\tilde{q'}) - r'(\tilde{q})$.

While $r$ and $r'$ have the same gradient $\Delta_r = \Delta_{r'}$, the values for $\mu_r$ and $\mu_{r'}$ may differ, i.e. the places of a Petri net constructed from these regions may have different self-loops.

By definition, there are only a finite number of $k$-bounded regions. We can use all of them and define a Petri net:

Definition 9. For a finite LTS $A$ and a number $k \in \mathbb{N}^+$, let $SN_k(A)$ be the Petri net generated from all $k$-bounded regions of $A$.

---

1 We exclude infinite LTS since they lead to unbounded Petri nets, for which the synthesis from modal transition systems is known to be undecidable [14].
An example for this construction with $k = 1$ is shown in Figure 2. The LTS $\mathcal{A}$ has only four regions $r_1$ to $r_4$. We identify a region $r$ with the vector $(r(q_0), r(q_1), r(q_2), r(q_3))$:

$$
r_1 = (0, 0, 0, 0) \quad r_2 = (0, 0, 0, 1) \quad r_3 = (1, 1, 0, 0) \quad r_4 = (1, 1, 1, 1)
$$

We also identify the gradient $\Delta_r$ and minimum values $\mu_r$ of $r$ with the vector $(\Delta_r(a), \Delta_r(b))$, respectively $(\mu_r(a), \mu_r(b))$:

$$
\Delta_{r_1} = (0, 0) \quad \Delta_{r_2} = (0, 1) \quad \Delta_{r_3} = (0, -1) \quad \Delta_{r_4} = (0, 0)
\quad \mu_{r_1} = (0, 0) \quad \mu_{r_2} = (0, 0) \quad \mu_{r_3} = (1, 1) \quad \mu_{r_4} = (1, 1)
$$

The event $a$ always has a gradient $\Delta_r(a) = 0$, since otherwise at least one of the values $r(q_0), r(q_1), r(q_2)$ would need to be outside of $\{0, 1\}$. For the event $b$, we can have $\Delta_r(b) \in \{-1, 0, 1\}$. Altogether, $\mathcal{A}$ has four regions which correspond to the places of the Petri net shown in the middle of Figure 2. As can be seen in the reachability graph, the event $a$ forms a loop around the initial state.

**Lemma 10.** For any $k \in \mathbb{N}^+$ and finite LTS $\mathcal{A}$, the Petri net $\text{SN}_k(\mathcal{A})$ is $k$-bounded.

**Proof.** Assume a reachable marking $M$ where some place $p$ contains more than $k$ tokens, i.e. $M(p) > k$. We use the well-known idea of complement places: With $r$ being the region corresponding to $p$, define a new region $r'$ via $r'(q) = k - r(q)$. Clearly, $r'$ is also a region of $\mathcal{A}$ and corresponds, by definition, to some place $p'$ of $\text{SN}_k(\mathcal{A})$, which is built from all regions of $\mathcal{A}$. Also, for every event $a \in \Sigma_\mathcal{A}$ we have $\Delta_r(a) + \Delta_r'(a) = 0$ and for every state $q$ we have $r(q) + r'(q) = k$. This also holds for the corresponding places: for any reachable marking $M', M'(p) + M'(p') = k$. Since $M(p) > k$ we find $M(p') < 0$. This contradicts $M$ being a (reachable) marking.

Next, we want to show that $\text{SN}_k(\mathcal{A})$ is the smallest Petri net over-approximation of $\mathcal{A}$. This means that any other Petri net which is an over-approximation of $\mathcal{A}$ will also be an over-approximation of the reachability graph of $\text{SN}_k(\mathcal{A})$.

**Theorem 11.** Given a number $k \in \mathbb{N}^+$ and a finite LTS $\mathcal{A}$, we have that $\mathcal{A} \sqsubseteq \text{RG}(\text{SN}_k(\mathcal{A}))$ and for all $k$-bounded Petri nets $\mathcal{N}$ with $\mathcal{A} \sqsubseteq \text{RG}(\mathcal{N})$, also $\text{RG}(\text{SN}_k(\mathcal{A})) \sqsubseteq \text{RG}(\mathcal{N})$.

**Proof.** It is clear that $\mathcal{A} \sqsubseteq \text{RG}(\text{SN}_k(\mathcal{A}))$, because $\text{SN}_k(\mathcal{A})$ is defined in such a way that it does not prevent edges present in $\mathcal{A}$ ($q \overset{t}{\rightarrow} \Rightarrow r(q) \geq \mu_r(t) = F(p, t)$).

Then, for any $\mathcal{N}$ with $\mathcal{A} \sqsubseteq \text{RG}(\mathcal{N})$ via $f$, each place of $\mathcal{N}$ corresponds to a region $r$ on $\text{RG}(\mathcal{N})$. This region can be translated into a region $r \circ f$ of $\mathcal{A}$ via Lemma 8, which by
definition corresponds to a place of $\text{SN}_k(A)$, yielding a mapping of places\(^2\) $\alpha: P_N \rightarrow P_{\text{SN}_k(A)}$.

We can now define a function $\beta$ that maps markings of $\text{SN}_k(A)$ to markings of $N$ via $\beta(M)(p) = M(\alpha(p))$, i.e., a place $p$ of $N$ is assigned the same number of tokens as its corresponding place $\alpha(p)$ of $\text{SN}_k(A)$ has in $M$. We can easily verify $M[t]M' \Rightarrow \beta(M)[t]\beta(M')$, i.e., we have $\text{RG}(\text{SN}_k(A)) \subseteq \text{RG}(N)$ via $\beta$.

\section{Modal Transition Systems}

Hyper modal transition systems\(^{18}\) represent classes of LTS. They allow for multiple initial states and have two different classes of edges. May edges allow an action while must edges require an action. Compared to classical modal transition systems\(^{17}\), must edges are hyper edges that require at least one out of several possible actions, while in a standard modal transition system no such logical combinations are expressible. A hyper modal transition system is similar to an acceptance specification\(^{19}\), but its must edges can be considered to be in conjunctive normal form while an acceptance specification uses the disjunctive normal form.

\begin{definition}
A hyper modal transition system (hMTS) $M$ is a structure $M = (S, \Sigma, \rightarrow, \ldots, S_0)$ where $S$ is a finite set of states, $\Sigma$ is a set of actions, $\emptyset \neq S_0 \subseteq S$ is the set of initial states, $\ldots \subseteq S \times \Sigma \times S$ is the set of may edges and $\ldots \subseteq S \times (2^{\Sigma \times S} \setminus \emptyset)$ is the set of must edges satisfying $\forall (s, D) \in \rightarrow: \forall (a, s') \in D: (s, a, s') \in \ldots$. The syntax $s \rightarrow^w s'$ is defined just like $q \rightarrow^w q'$ was defined for LTS. An hMTS is called deterministic iff $S_0 = \{s_0\}$ is a singleton and the LTS $(S, \Sigma, \ldots, s_0)$ is deterministic. A modal transition system (MTS) is an hMTS with $|S_0| = 1$ and $\forall (s, D) \in \rightarrow: |D| = 1$.
\end{definition}

As an example, three hMTSs are shown in Figure 3. In $M_1$, the lower path forms a $bac$-loop that can be terminated after an $a$, the upper path requires $abc$, after which anything is allowed. $M_2$ contains two hyper edges requiring at least either an $a$ or a $b$, but not necessarily both. In an LTS meeting the specification of $M_3$, the three must edges at $s_0$ demand that both $a$ and $b$ must be available actions, while the hyper edge at $s_1$ only requires that one of $a$ and $b$ is possible.

The following definition introduces how an LTS implements an hMTS. This implementation relation is similar to a bisimulation, but may and must edges are not treated the same.

\(^2\) A mapped place $\alpha(p)$ has the same initial marking and transitions of $\text{SN}_k(A)$ have the same effect $\Delta_\alpha = \Delta_{\alpha(p)}$ on it as on $p$, but self-loops might be added (if $\mu_{\alpha(t)} > \mu_t$ due to edges not existing in $\text{RG}(\text{SN}_k(A))$ but in $\text{RG}(N)$).
A must edge specifies a disjunction of edges, at least one of which must be present, while a may edge defines that some edge is allowed.

**Definition 13.** An LTS \( A \) is an implementation via \( R \subseteq Q_A \times S_M \) of an hMTS \( M \) if there is an \( s_0 \in S_{0,M} \) with \( (q_0, A, s_0) \in R \) and for all \( (q, s) \in R \):

\[
\forall q \xrightarrow{a} q' : \exists s \xrightarrow{a} s' : (q', s') \in R,
\]

\[
\forall (s, D) \in \xrightarrow{-} : \exists (a', s') \in D, q \xrightarrow{a'} q' : (q', s') \in R.
\]

If \( A \) is an implementation of \( M \) via some relation \( R \), we call \( R \) an implementation relation and write \( A \models M \) (via \( R \)). If \( RG(N) \) is an implementation of \( M \) for some Petri net \( N \), we also say that \( N \) realises \( M \).

Examples for this definition are given in Figure 4. The LTS \( A_1 \) is an implementation of \( M_1 \) from Figure 3. An implementation relation \( R \) can be built as follows. Initially, \( (\{s_0, s_3\}, s_0) \in R \). The lower path of \( M_1 \) adds first \( (\{s_3, s_4\}, s_4) \) and then \( (\{s_2, s_3, s_5\}, s_5) \) to \( R \). The must edges of the upper path require \( (\{s_1, s_3\}, s_1) \), \( (\{s_2, s_3, s_5\}, s_2) \), and \( (\{s_0, s_3\}, s_3) \). In the state \( s_3 \), all edges are allowed and loop back to \( s_3 \), i.e., all states reachable from \( \{s_0, s_3\} \) must also be related to \( s_3 \): \( (\{s_1, s_3\}, s_3) \), \( (\{s_3, s_4\}, s_3) \), \( (\{s_2, s_3, s_5\}, s_3) \) \( \in R \). Thus, the state names of the LTS \( A_1 \) show exactly the implementation relation for \( A_1 \models M_1 \). If we remove in \( M_1 \) the may edge labelled with \( c \) at \( s_5 \), \( A_1 \) would not be an implementation any more: \( (\{s_2, s_3, s_5\}) \) must admit a \( c \) (forced by the upper path in \( M_1 \)) and forbid a \( c \) (using the lower path) at the same time. We could split up the state \( \{s_2, s_3, s_5\} \) into \( \{s_2, s_3\} \) and \( \{s_5\} \), but then no Petri net could solve the LTS, since the paths \( ab \) and \( ba \) from the initial state must lead to the same marking\(^3\).

The four LTS in the middle of Figure 4 are all valid implementations of \( M_2 \), and they are all minimal according to the LTS homomorphism \( \sqsubseteq \), which is a partial order for deterministic and totally reachable LTS according to Lemma 3. There are larger implementations admitting \( a \) and \( b \) at either \( s_0 \) or \( s_1 \) (or both).

The LTS \( A_3 \) is a minimal implementation of \( M_3 \). The states \( s_1, s_3, \) and \( s_5 \) are forced by the implementation relation; the hyper edge at \( s_1 \) allows a choice: at least one of the targets of the hyper edge must exist. Here, we chose \( s_2 \). Note that the states \( s_1 \) and \( s_3 \) cannot be identified in spite of the non-deterministic \( a \)-edges with the same source \( s_0 \). The implementation relation would contain a pair \( (\{s_1, s_3\}, s_3) \) which would in consequence – due to the edge \( (\{s_1, s_3\}) \xrightarrow{a} s_2 \) – require \( s_3 \) to have at least a may edge in \( M_3 \). Since reachability graphs of Petri nets are deterministic, no LTS implementing \( M_3 \) can be solved by a Petri

\(^3\) This is due to the well-known Petri net state equation \( M[w]M' = M' = M + C \cdot \Psi(w) \) where \( C \in \mathbb{Z}^{P \times N} \) with \( C(p, t) = F(t, p) - F(p, t) \). This equation only considers the number of times \( \Psi(w)(t) \) that event \( t \in T \) occurs in the word \( w \in T^* \), and not the order of appearances.
net. However, if we omit the edge \((s_0, \{(a, s_3)\}) \in \longrightarrow \) in \(M_3\), Petri net realisations become possible. There are (up to isomorphism) nine different deterministic LTSs implementing the modified \(M_3\), five of which can be solved by Petri nets. They differ in whether \(a\), \(b\), or both are possible in \(s_1\), and in the number of states without outgoing edges.

4 Goal-oriented Synthesis

In this section, we will introduce the algorithm that calculates Petri net realisations for hyper modal transition systems. The algorithm works by iteratively adding unimplemented must edges to an LTS and doing minimal Petri net over-approximations. For identifying unimplemented must edges, another kind of relation is needed, which we will define next.

Definition 14. Let \(A\) be an LTS and \(M\) be an hMTS. A relation \(R \subseteq Q_A \times S_M\) is called an expansion relation if there is a state \(s_0 \in S_{0,M}\) with \((q_{0,A}, s_0) \in R\) and for all \((q, s) \in R\) and all \(q \xrightarrow{a} q'\), there is an \(s'\) with \(s \xrightarrow{a} s'\) and \((q', s') \in R\).

Let \(R_E(A, M)\) be the set of all expansion relations of \(A\) and \(M\). Given LTS \(A\) and \(B\), an expansion relation \(R \in R_E(A, M)\), and an LTS homomorphism \(f\) witnessing \(A \sqsubseteq B\), the relation \(f(R)\) is defined as \(f(R) = \{(f(q), s) \in Q_B \times S_M \mid (q, s) \in R\}\).

Expansion relations are a weaker variant of implementation relations, i.e. any implementation relation is an expansion relation with an additional condition for the must edges.

Our algorithm for realising an hMTS is Algorithm 1. The idea behind the algorithm is to iteratively enlarge a current LTS towards becoming an implementation. The first operation for this is to add edges to the LTS so that currently unimplemented must edges become implemented (EXPAND). The second operation is the minimal Petri net over-approximation (PNAPPROX). The algorithm begins in the procedure REALISEMTS, which gets as input the bound \(k \in \mathbb{N^+}\) for the Petri nets and the hMTS \(M\) to be realised. The first LTS to be considered is the minimal LTS which only has an initial state and no edges. This state is related to each possible initial state of \(M\) and the procedure RECURSE is called for each such combination.

RECURSE identifies unimplemented must edges by negating the corresponding formula in the definition of an implementation relation. If no must edge is missing, a solution is found and returned. Otherwise, the procedure EXPAND is invoked which will add new states and edges to implement the missing must edges. This procedure returns a set, because, for example, for \(m = \{(q, \{(a, s_1), (b, s_2)\})\}\), two LTSs are created: In one of them, an edge \(q \xrightarrow{a} q_{\text{new}}\) is created, while the other one gets the new edge \(q \xrightarrow{b} q_{\text{new}}\). In the first case, \((q_{\text{new}}, s_1)\) is added to the expansion relation \(R_A\) while in the second case \((q_{\text{new}}, s_2)\) is added. This means the relation \(R_A\) is enlarged to keep note on which state of the hMTS the new state should implement. EXPAND is invoked recursively to obtain all combinations of selections for the missing must edges. In MTS must edges allow no choice, therefore EXPAND will return one unique result; in hMTS a high number of choices in must edges could lead to an exponential growth, though.

For each result of EXPAND, the procedure PNAPPROX is called, yielding a minimal Petri net over-approximation of its argument. As was seen in Figure 2, this can greatly influence the shape of the LTS under consideration. The relation \(R_A\) has to be updated to this new LTS. To do so, PNAPPROX computes a homomorphism witnessing \(A \sqsubseteq B\) in line 23. Such a homomorphism exists by Theorem 11. It is unique by Lemma 2, because \(A\) is totally reachable by construction and \(B\) is deterministic by Lemma 5. This homomorphism \(f\) is then used to produce the relation \(f(R_A)\) which relates \(B\) to \(M\). The set \(E\) of all expansion relations
Algorithm 1 Algorithm for finding Petri net realisations for an hMTS.

1: procedure RealisEMTS(k, M) \( \triangleright k \in \mathbb{N}^+ \) and M is an hMTS
2: Let A be the LTS consisting of just an initial state \( q_0 \)
3: return \( \bigcup_{s_0 \in S_0,M} \text{RECURSE}(k, A, \{(q_0, s_0)\}, M) \)
4: end procedure

5: procedure Recurse(k, A, Ra, M) \( \triangleright \) Ra is an expansion relation
6: \( m = \bigcup_{(q,s) \in Ra} m_{q,s} \) where \( m \) collects missing must edges
7: \( m_{q,s} = \{ (q, D) \in QA \times (2^S \times SM) \setminus \emptyset \} \mid (s, D) \in \rightarrow_M, \forall (a, s') \in D: \neg \exists q' \in QA: (q \xrightarrow{a} q' \land (q', s') \in RA) \}
8: if \( m = \emptyset \) then return \( \{(A, Ra)\} \) end if
9: return \( \bigcup_{(B, Ra) \in \text{EXPAND}(A, Ra, m)} \bigcup_{(C, Rc) \in \text{PNAPPROX}(k, B, Ra, M)} \text{RECURSE}(k, C, Rc, M) \)
10: end procedure

11: procedure Expand(A, Ra, m)
12: if \( m = \emptyset \) then return \( \{(A, Ra)\} \) end if
13: Select some \( (q, D) \in m \), add a new state \( q_{\text{new}} \) to A and set result \( \leftarrow \emptyset \)
14: for \( (a, s') \in D \) do \( \triangleright \) Implement missing must edge
15: Let B be a copy of A with an additional edge \( q \xrightarrow{a} q_{\text{new}} \)
16: result \( \leftarrow \) result \( \cup \) Expand(B, Ra, \( \{q_{\text{new}}, s'\}\), m \( \setminus \) \( \{(q, D)\}\))
17: end for
18: return result
19: end procedure

20: procedure PNAPPROX(k, A, Ra, M)
21: \( B = RG(SN_k(A)) \) \( \triangleright \) Minimal over-approximation
22: \( f = \) homomorphism witnessing \( A \subseteq B \)
23: \( E = \{ R_B \in RE(B, M) \mid f(Ra) \subseteq R_B \} \) \( \triangleright \) All expansion relations containing \( f(Ra) \)
24: return \( \{(B, R_B) \mid R_B \in E \land \neg \exists R' \in E: R' \subsetneq R_B \} \) \( \triangleright \) Select minimal relations
25: end procedure

containing \( f(Ra) \) is computed and the minimal relations are returned. It is possible that this produces several different minimal relations, or none at all. In the latter case, PNAPPROX returns the empty set and this branch of the algorithm fails to find any realisation of the hMTS M.

4.1 Examples

As our first example, we will use \( M_1 \) from Figure 3 and realise it with a 1-bounded Petri net. The call RealisEMTS(1, \( M_1 \)) begins by creating an LTS \( A \) consisting of just an initial state \( q_0 \) which gets related to the initial state \( s_0 \) of \( M_1 \). In Recurse(1, A, \( \{(q_0, s_0)\}, M_1 \)), the set \( m = \{(q_0, \{(a, s_1)\}), (q_0, \{(b, s_1)\})\} \) is calculated. Therefore, \( q_0 \) needs an outgoing edge with label \( a \) and another edge with label \( b \). The invocation Expand(A, \( \{q_0, s_0\}\), m) produces just the pair \( (A_1, R_1) \) where \( A_1 \) is shown in Figure 5. The same figure also visualises \( R_1 \) by labelling a state \( q \) in the LTS with \( s \) if \( (q, s) \in R_1 \). The following call to PNAPPROX does not modify this LTS and Recurse calls itself recursively with \( (A_1, R_1) \).

The next call to Expand produces the LTS \( A_1 \) from Figure 5. This LTS cannot be solved by a Petri net, because if both sequences \( ab \) and \( ba \) are enabled in a marking, they must lead to the same state. Thus, PNAPPROX produces the LTS \( A_2 \). The relation that is indicated by its labelling is \( f(R_1) \), i.e. no additional entries need to be added to produce an expansion relation.

This expansion procedure continues for some iterations until the LTS \( A_5 \) from Figure 5
Figure 5 Intermediate LTS from the Petri net synthesis of the deterministic MTS $M_1$ from Figure 3. Some intermediate results are shown. To visualise a relation $R$, the states of an LTS are labelled with the states of $M_1$ that they have to implement.

is created by EXPAND. This LTS cannot be solved by a 1-bounded Petri net. As we have already seen in Figure 2, when some firing sequence $ww \in \Sigma^*$ is possible in a 1-bounded Petri net, then it has twice the effect of the sequence $w$ on the marking, i.e. it cannot have an effect at all. Thus, it forms a loop in the reachability graph. The same thing happens now with the sequence $abc$. The minimal over-approximation produces the LTS $A_6$. The relation returned by $\text{PNApprox}$ is just $R_6 = f(\tilde{R}_5)$. At this point all must edges are implemented and we have found a realisation.

As another example, consider $\text{RealiseMTS}(2, M_2)$ where $M_2$ is shown in Figure 3. This produces the four LTS that are displayed in the middle of Figure 4.

If one considers $\text{RealiseMTS}(1, M_2)$ instead, i.e. $k = 1$ instead of $k = 2$, then only the LTS corresponding to the words $ab$ and $ba$ are generated. This is because, for example, $aa$ cannot be generated by one-bounded Petri nets without also allowing $aaa$ to occur. This extra behaviour is not allowed by $M_2$ and so the algorithm will end up with $E = \emptyset$ in line 24 and this $aa$-branch of the recursion does not find any realisations.

When applying the algorithm to $M_3$ of Figure 3, we get an LTS with states corresponding to $s_0$, $s_1$, $s_3$, and $s_5$ in the first expansion phase. The over-approximation will then combine $s_1$ and $s_3$ to $\{s_1, s_3\}$ since both $a$-edges must have the same effect on the marking of the Petri net. In the second expansion phase, one of the edges to $s_4$, $s_2$, or $s_6$ is added to the LTS, yielding three LTS to over-approximate. While the over-approximations exist, no expansion relation can be built for them. For the pair $\{(s_1, s_3), \}$, the LTS state $\{s_1, s_3\}$ has an edge now (the newly added $a$ or $b$), but $s_3$ in $M_3$ has no may edge at all. Thus, $E = \emptyset$ and $\text{PNApprox}$ returns the empty set in all cases. The realisation of $M_3$ by a Petri net has failed (correctly).

5 Termination and Correctness

In this section, three results will be shown: The algorithm terminates; the algorithm is correct in the sense that it really produces realisations of the hMTS $M$; and the algorithm is complete in the sense that it always finds a realisation if one exists.

Lemma 15. For an hMTS $M$ and a $k \in \mathbb{N}^+$, a call to $\text{RealiseMTS}(k, M)$ terminates.
Proof. If the algorithm does not terminate, it must be due to \textsc{Recurse} because PNApprox contains only finite constructions and \textsc{Expand} has a recursion depth of $|m|$. Since the unions in line 10 of the algorithm are always finite, there must be an infinite recursion of \textsc{Recurse} with arguments $(A_i)_{i \in \mathbb{N}}$. All arguments to \textsc{Recurse} are Petri net solvable by construction. Also, by Lemma 6, there are only finitely many LTS (abstracting state names) over a fixed alphabet which are $k$-bounded Petri net solvable. Thus, the $A_i$ are all elements of a finite set and form an infinite chain with $A_i \subseteq A_{i+1}$ for $i \in \mathbb{N}$, i.e. some LTS $A$ (up to isomorphism) will occur infinitely often. Let $A_j$ and $A_j'$ be any two instances of $A$ with $j < j'$, so $A_j = A_j'$. For $j < n < j'$ then $A_j \subseteq A_n \subseteq A_{j'}$ and Lemma 3 can be applied since all $A_i$ are reachability graphs and thus by Lemma 5 totally reachable and deterministic. We conclude $A_j = A_n$ (up to isomorphism), i.e. the infinite chain $(A_i)_{i \in \mathbb{N}}$ becomes stationary.

Consider now some $i \geq j$. Let $\tilde{A}_i$ be the LTS resulting from applying \textsc{Expand} to $A_i$, possibly adding one (or more) states $q_{\text{new}}$ to $A_i$. Then, $A_i \subseteq \tilde{A}_i$, via the identity homomorphism $id$, where the new state(s) $q_{\text{new}}$ do not occur as an image. Now, PNApprox recreates the original LTS, i.e. $A_i \subseteq \tilde{A}_{i+1} = A_i$ via some homomorphism $f$. Overall, $A_i \subseteq \tilde{A}_i \subseteq A_i$ via $f \circ id$, which is uniquely determined according to Lemma 2. Thus, $f \circ id$ must be the identity mapping, and $f$ also must be the identity on all states except the new one(s), $q_{\text{new}}$.

If \textsc{Expand} would add an edge $q \xrightarrow{a} q_{\text{new}}$ to $A_i$, then $f(q) \xrightarrow{a} f(q_{\text{new}})$ would also be an edge in $A_i = A_{i+1}$. PNApprox maps $f(q) = q$ and $f(q_{\text{new}}) = q' \in Q_A$. The new element of the expansion relation added by \textsc{Expand}, $(q_{\text{new}}, \tilde{s})$ where $\tilde{s}$ is determined by $M$, is mapped to $(q', \tilde{s}) \in R_{A_{i+1}}$. Note that $(q', \tilde{s}) \notin R_{A_i}$, otherwise \textsc{Expand} would not have added an edge in the first place. Therefore, $|R_{A_i}| < |R_{A_{i+1}}|$. Since the chain $(A_i)_{i \geq j}$ is stationary, the maximal size of an expansion relation is $|Q_A \times S_M|$, i.e. at some point in the chain, \textsc{Expand} cannot add any further edges. Therefore, the to-be-implemented set $m$ will be empty and \textsc{Recurse} terminates.

The following lemma shows that \textsc{RealiseMTS} only produces realisations of its input:

\textbf{Lemma 16.} For an hMTS $M$ and a $k \in \mathbb{N}^+$, every $(A, R_A) \in \text{RealiseMTS}(k, M)$ satisfies $A \models M$ via $R_A$ and $A$ can be solved by a $k$-bounded Petri net.

Proof. For $(A, R_A)$ to appear in the result of \textsc{RealiseMTS}, it must satisfy $m = \emptyset$ in line 9 of the algorithm. This means that there are no unimplemented must edges, which shows the first half of $A \models M$ via $R_A$. The other half, that all edges in $A$ are allowed by may-edges, holds, because every value of $R_A$ in the algorithm is an expansion relation: Initially, $A$ only has an initial state and $R_A$ relates this state to an initial state of $M$. When \textsc{Expand} adds new edges to an LTS, the relation $R_A$ is updated accordingly. Here, $(s, a, s')$ is a may edge, because every must edge must also be a may edge by definition of an hMTS. Finally, PNApprox explicitly only returns expansion relations.

It remains to be shown that $A$ can be solved by a $k$-bounded Petri net. This is the case, because every LTS $A$ given to \textsc{Recurse} is either the trivial LTS having no edges or was generated by PNApprox as a minimal Petri net over-approximation.

In the remainder of the section, we will show that \textsc{RealiseMTS}(k, M) = $\emptyset$ can only occur if there are no realisations of $M$. For this we need the following preorder on LTS enriched with a relation:

\textbf{Definition 17.} Let $S$ be an arbitrary set. For two LTS $A$ and $B$ assume relations $R_A \subseteq Q_A \times S$ and $R_B \subseteq Q_B \times S$. We write $(A, R_A) \sqsubset (B, R_B)$ if there is an LTS
homomorphism $f$ witnessing $A \subseteq B$ so that $\forall (q, s) \in R_A : (f(q), s) \in R_B$, i.e. $f(R_A) \subseteq R_B$ (cf. Definition 14).

**Lemma 18.** Let $M$ be an $hMTS$, $k \in \mathbb{N}^+$, and $N$ be a Petri net with $RG(N) \models M$ via some relation $R_N$. There is an $(A, R_A) \in REALISEMTS(k, M)$ so that $(A, R_A) \subseteq (RG(N), R_N)$.

**Proof.** We prove inductively that there is always an $(A, R_A)$ in the current state of the algorithm that satisfies $(A, R_A) \subseteq (RG(N), R_N)$.

By $RG(N) \models M$ via $R_N$, there must be an $s_0 \in S_{0,M}$ with $(M_0, s_0) \in R_N$ ($M_0$ being the initial marking of $N$). Since REALISEMTS tries all initial states of $M$, the same $s_0$ will be picked at some point, i.e. $\text{RECURSE}(k, A, R_A, M)$ will be called with $A$ being the LTS having just a single state and the relation $R_A = \{(q_0, s_0)\}$. Here, $(A, R_A) \subseteq (RG(N), R_N)$ holds.

Next, assume that $\text{RECURSE}$ is called with arguments $A$ and $R_A$ satisfying $(A, R_A) \subseteq (RG(N), R_N)$ via some homomorphism $f$. We have to show two things: (1) One of the pairs $(B, R_B)$ generated by $\text{EXPAND}(A, R_A, m)$ satisfies $(B, R_B) \subseteq (RG(N), R_N)$ and (2) the same applies to one of the pairs $(C, R_C)$ which are generated by the following call of $\text{PNAPPROX}(k, B, R_B, M)$ in line 10. Since the algorithm always terminates by Lemma 15, eventually $m = \emptyset$ in line 9 will cause a suitable $(A, R_A)$ to be returned. All recursive calls to $\text{RECURSE}$ pass this on and in the end the pair $(A, R_A)$ is returned by $\text{REALISEMTS}$, completing the proof.

For (1) consider any unimplemented $(q, D) \in m$. By construction of $m$ there must be a $(q, s) \in R_A$ so that $(q, D) \in m_{q,s}$ and $(s, D) \in \rightarrow_M$. By $(A, R_A) \subseteq (RG(N), R_N)$ via $f$, we have $(f(q), s) \in R_N$. By $(s, D) \in \rightarrow_M$, $(f(q), s) \in R_N$ and $RG(N) \models M$ via $R_N$, there must be some $(a, s') \in D$ and $\tilde{M}$ with $f(q) \xrightarrow{a} \tilde{M}$ and $(\tilde{M}, s') \in R_N$. Thus, on our way to $B$ we can pick that LTS generated in the iteration of line 15 of the algorithm which handles the same $(a, s') \in D$. We define a new function $f'$ that is identical to $f$, except that its domain is extended by $q_{\text{new}}$ via $f'(q_{\text{new}}) = \tilde{M}$. Having done this for all unimplemented must edges, the final function $f'$ witnesses $(B, R_B) \subseteq (RG(N), R_N)$: Most of this property is inherited from $f$ and for any new edge $q \xrightarrow{a} q_{\text{new}}$, we constructed $f'$ so that $f'(q) \xrightarrow{a} f'(q_{\text{new}})$ is satisfied.

For (2) we assume that we have $(B, R_B) \subseteq (RG(N), R_N)$ via some function $f_B$. We have to find some $(C, R_C) \in \text{PNAPPROX}(k, B, R_B, M)$ with $(C, R_C) \subseteq (RG(N), R_N)$. The LTS relations are illustrated in Figure 6. From the algorithm we know $C = RG(SN_k(B))$. From the assumption we get $B \sqsubseteq RG(N)$ via $f_B$. Applying Theorem 11 to $B$, we can deduce $C = RG(SN_k(B)) \sqsubseteq RG(N)$ via some function $f_C$ and $B \sqsubseteq C$ via some function $f$. We have $B \sqsubseteq C \sqsubseteq RG(N)$ via $f_C \circ f$ by transitivity of $\sqsubseteq$. Since $B$ is totally reachable by construction and $RG(N)$ is deterministic by Lemma 5, we can apply Lemma 2 stating that the homomorphism witnessing $B \sqsubseteq RG(N)$ is unique. We conclude $f_B = f_C \circ f$.

Consider the expansion relation $R_C$ defined by $R_C = \{(q, s) \in Q_C \times S_M \mid (f_C(q), s) \in R_N\}$. This relation inherits the expansion property from $R_N$, i.e. if $(q, s) \in R_C$ and $(q, a, q') \in \rightarrow_C$, then $(f_C(q), s) \in R_N$ and $(f_C(q), a, f_C(q')) \in \rightarrow_{RG(N)}$, which means that there is a suitable $s' \in S_M$ with $s \xrightarrow{a} s'$ and $(f_C(q'), s') \in R_N$, so $(q', s') \in R_C$. By definition we have $(C, R_C) \subseteq (RG(N), R_N)$ via $f_C$. Consider any element $(f(q), s) \in f(B)$. By definition of $f(R_B)$, we have $(q, s) \in R_B$. By $(B, R_B) \subseteq (RG(N), R_N)$ via $f_B$, we have
Figure 7 An LTS that can be solved exactly, but only if non-minimal regions are considered.

$(f_B(q), s) \in R_N$. By $f_B = f_C \circ f$, we have $(f_C(f(q)), s) \in R_N$. By definition of $R'_C$, we have $(f(q), s) \in R'_C$. Thus, $f(R_B) \subseteq R'_C$, which means that $R'_C$ is an element of the set $E$ inside the algorithm, i.e. there is a relation $R_C \subseteq E$ with $f(R_B) \subseteq R_C \subseteq R'_C$ so that $(C, R_C) \in \text{PNAPPROX}(k, B, R_B, M)$. Since $(C, R_C) \subseteq (\text{RG}(N), R_N)$ via $f_C$ and $R_C \subseteq R'_C$, we also have $(C, R_C) \subseteq (\text{RG}(N), R_N)$ by the definition of $\subseteq$, which needed to be shown.

**Corollary 19.** For an hMTS $M$ and a $k \in \mathbb{N}^+$, if $\text{RealiseMTS}(k, M) = \emptyset$ then there is no $k$-bounded Petri net $N$ with $\text{RG}(N) \models M$.

### 6 Some final thoughts

An aspect of the algorithm that is still open is its complexity. This depends a lot on the complexity of the Petri net over-approximation. The question of whether for a given LTS $A$ any 1-bounded and pure Petri net $N$ exists with $A = \text{RG}(N)$ is NP-complete [4]. The same question can be answered in polynomial time when asking for bounded Petri nets, but without picking a bound $k \in \mathbb{N}^+$ a priori [3]. We do not know about any complexity results for the synthesis of $k$-bounded Petri nets with a given $k$, nor for the $k$-bounded over-approximation that is needed in our present setting.

When looking at some algorithms for this problem, we can see that the brute force computation of all regions will certainly be exponential (checking $(k + 1)^{|Q|}$ functions) in the size of the LTS, but not all regions might be necessary. Attempts to reduce this problem have been made, e.g. in [10, 11], but this procedure does not guarantee success. It only computes minimal regions, where a region $r$ is **minimal** if there is no other region $r' \neq 0$ with $r' \leq r$ (pointwise). Figure 7 contains an LTS which cannot be solved when just using minimal regions. The following list contains all six minimal regions of this LTS, where a region $r$ is identified with the vector $(r(q_0), r(q_1), \ldots, r(q_6))$ and a gradient $\Delta_r$ with the vector $(\Delta_r(a), \Delta_r(b))$:

$r_1 = (1, 0, 1, 2, 1, 0, 0) \quad \Delta_{r_1} = (-1, 1) \quad r_2 = (2, 2, 1, 0, 0, 0, 1) \quad \Delta_{r_2} = (0, -1)$

$r_3 = (1, 2, 1, 0, 1, 2, 2) \quad \Delta_{r_3} = (1, -1) \quad r_4 = (0, 2, 1, 0, 2, 4, 3) \quad \Delta_{r_4} = (2, -1)$

$r_5 = (0, 1, 1, 1, 2, 3, 2) \quad \Delta_{r_5} = (1, 0) \quad r_6 = (0, 0, 1, 2, 2, 1) \quad \Delta_{r_6} = (0, 1)$

It can easily be verified that for all of these regions we have $r(q_0) \geq \mu(r, b) = F(r, b)$, which means that none of these regions prevents transition $b$ in state $q_0$. However, the region $r_7 = (3, 2, 2, 2, 1, 0, 1)$ does prevent $b$ in $q_6$ ($r_7(q_6) = 1 < 2 = \mu_{r_7}(b)$). This region is not minimal since $r_1 \leq r_7$, but only with this region can the LTS from Figure 7 be solved.

In the general setting of bounded Petri net synthesis without $k$ given a priori, there are polynomial algorithms based on solving so-called separation problems (see e.g. [3, 5]). These algorithms seem to be quite efficient in our experiments, but so far there do not seem to be variants where $k$ is given a priori nor for computing a minimal over-approximation. If further research finds such an algorithm, another optimisation becomes possible: When a must edge is present, the corresponding event does not need to be prevented even if this is
possible. Thus, instead of computing the minimal over-approximation, the EXPAND-step could be integrated with the synthesis to make it faster.

Coming back to the problem of finding a Petri net realisation of an MTS, the brute force approach would be the construction of all \(k\)-bounded Petri nets with a fixed set of transitions. This also requires the computation of all possible regions (places), but afterwards we have to walk through the power set of the places, construct the according reachability graphs, and check if they implement our MTS. As long as the Petri net over-approximation is not used too often, our algorithm will probably be faster.

Still, EXPAND and PNApprox are independent procedures, so it might be feasible to reduce the number of over-approximations in favour of the EXPAND step. We can modify our algorithm, doing as many consecutive EXPAND steps as possible (e.g. until a cycle in the MTS becomes fully implemented) and only then applying one over-approximation.

Of course, with general hMTS, the EXPAND step can itself create exponentially many LTS, but we might expect that true hyper edges (defining algorithmic points of choice) occur with low frequency in at least human-made hMTS. For MTS (without true hyper edges), every EXPAND step only computes a single LTS. If an hMTS is deterministic, it can easily be shown that there is at most one expansion relation and so PNApprox would also compute at most a single result. Thus, the algorithm does not branch when given a deterministic MTS.

Overall, we obtain a goal-oriented algorithm and thus a decision procedure for the realisability of hMTS by \(k\)-bounded Petri nets with hopefully much lower run times than the brute force approach by enumeration of all candidate nets. We are already close to the barrier of undecidability here, since for bounded Petri nets (without the fixed \(k \in \mathbb{N}^+\)), the realisability of even MTS is known to be undecidable [20].

Our approach can easily handle restricted Petri net classes such as plain Petri nets (arc weights at most 1) or pure Petri nets (see Definition 4). Instead of constructing \(\text{SN}_k(A)\) from all \(k\)-bounded regions, simply only regions corresponding to plain, respectively pure, places are considered.

Further research effort can be put into investigating more expressive specifications than hMTS. We would like to have a similar algorithm for finding Petri net realisations of formulas of the modal \(\mu\)-calculus [15, 2]. The general approach of the algorithm would stay the same, but a suitable replacement for the EXPAND step is needed.

Acknowledgements. We would like to thank Eike Best and Valentin Spreckels for their help. Also, we are thankful to the anonymous reviewers for their helpful comments.

References

A Characterisation of Open Bisimilarity using an Intuitionistic Modal Logic

Ki Yung Ahn\textsuperscript{1}, Ross Horne\textsuperscript{2}, and Alwen Tiu\textsuperscript{3}

1 School of Computer Science and Engineering, Nanyang Technological University, Singapore
   yaki@ntu.edu.sg
2 School of Computer Science and Engineering, Nanyang Technological University, Singapore
   rhorne@ntu.edu.sg
3 School of Computer Science and Engineering, Nanyang Technological University, Singapore
   atiu@ntu.edu.sg

Abstract

Open bisimilarity is a strong bisimulation congruence for the π-calculus. In open bisimilarity, free names in processes are treated as variables that may be instantiated; in contrast to late bisimilarity where free names are constants. An established modal logic due to Milner, Parrow, and Walker characterises late bisimilarity, that is, two processes satisfy the same set of formulae if and only if they are bisimilar. We propose an intuitionistic variation of this modal logic and prove that it characterises open bisimilarity. The soundness proof is mechanised in Abella. The completeness proof provides an algorithm for generating distinguishing formulae, useful for explaining and certifying whenever processes are non-bisimilar.

1998 ACM Subject Classification F.4.1 Mathematical Logic

Keywords and phrases bisimulation, modal logic, intuitionistic logic

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2017.7

1 Introduction

In this work, we consider open bisimilarity\textsuperscript{[13]} which ensures processes equivalence under any context at any point in their execution. Open bisimulation is an appealing choice of equivalence for state-space reduction due to its lazy call-by-need approach to inputs, which makes it easier to automate\textsuperscript{[15]}. In such a call-by-need approach, a value received is only observed when it needs to be used. Furthermore, some process calculi have been shown to enjoy sound and complete algebraic characterisations with respect to open bisimilarity.

The fine algebraic properties of open bisimilarity may be desirable for some applications. For many applications, it is desirable to avoid a situation where an equivalence technique proves that two components are equivalent in a sandbox test environment, when, in fact, they are distinguishable when plugged into a larger system. More subtly, processes may change context during execution\textsuperscript{[10]}; for example, virtual machines migrate between devices, and replicas replace components at runtime to keep a system live in the face of unavoidable node

\textsuperscript{*} The authors receive support from MOE Tier 2 grant MOE2014-T2-2-076. The third author receives support from NTU Start Up grant M4081190.020.

© Ki Yung Ahn, Ross Horne, and Alwen Tiu; licensed under Creative Commons License CC-BY
28th International Conference on Concurrency Theory (CONCUR 2017).
Editors: Roland Meyer and Uwe Nestmann; Article No. 7; pp. 7:1–7:17
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany
failures. For some notions of observational equivalence two processes may be indistinguishable when executed in any context prescribed; however, if the same two processes execute a few steps and then are migrated to another context, then it is possible that, from that point, the processes can exhibit observably distinct behaviours.

Process equivalences for the π-calculus coarser than open bisimilarity are prone to limitations described above. For instance, late bisimilarity [8] is not a congruence, since it is not preserved by input prefixes. Furthermore, even if we take the greatest congruence relation contained in late bisimilarity, called late congruence, late congruence is no longer a bisimulation hence is not necessarily preserved during execution. These issues are remedied by open bisimilarity [13].

The problem we address is the nature of a modal logic characterising open bisimilarity, in the tradition pioneered by Hennessy and Milner [6]. A modal logic characterising a bisimulation should have the property that whenever two processes are not bisimilar there should exist a distinguishing formula in the modal logic that holds for one process, but not for the other process. Such distinguishing formulae are useful for explaining why two processes are not bisimilar. Modal logics characterising late bisimilarity and coarser bisimulations were developed early in the literature on the π-calculus, by Milner, Parrow and Walker [9].

A novelty of our modal logic characterising open bisimilarity, which we name OM, is that it is intuitionistic rather than classical. A non-classical feature of OM is that box and diamond modalities have independent interpretations, except in special cases such as $[\tau]a\land b$ which is equivalent to $\neg([\tau]a \lor \neg([\tau]b))$. In general, in OM it is rarely the case that box can be defined in terms of diamond and negation. This contrasts to a classical modal logic we would expect that $[\pi]\phi$ and $\neg([\pi]\phi)$ define equivalent formulae, however such de Morgan dualities do not hold for most OM formulae.

More profoundly, the law of excluded middle does not hold in OM. For example, the process $\pi b \parallel c(x)$ does not satisfy the formula $\langle [\tau]a \parallel x \parallel \neg([\tau]a \parallel x) \parallel \tau \rangle S$, that is, $\pi b \parallel c(x) \not\models \langle [\tau]a \parallel x \parallel \neg([\tau]a \parallel x) \parallel \tau \rangle S$. The failure of the formula above relies on the fact that we have not yet fixed whether $a = c$ or $a \neq c$, which amounts to the absence of the law of excluded middle for name equality, as observed in related work on logical encodings of open bisimilarity [16]. In open bisimulation, both $a$ and $c$ are variables that may or may not be instantiated with the same value.

As a further example, consider the following two processes:

$$R \triangleq \tau.(\pi b.a(x) + a(x) \parallel \pi b \parallel \tau \parallel \tau.(\pi b.c(x) + c(x) \parallel \pi b))$$

$$S \triangleq \tau + \tau.(\pi b \parallel c(x))$$

The above processes are not open bisimilar. Process $R$ satisfies $[\tau]([\tau]a \parallel x \parallel \neg([\tau]a \parallel x) \parallel \tau)$ but process $S$ does not, since there is a $\tau$-transition to process $\pi b \parallel c(x)$ that we just agreed above does not satisfy $\langle [\tau]a \parallel x \parallel \neg([\tau]a \parallel x) \parallel \tau \rangle$. In this example, the absence of the law of excluded middle is necessary for the existence of a formula distinguishing these processes in OM.

The absence of de Morgan dualities discussed above complicates the construction of distinguishing formulae for processes that are not open bisimilar. For example, $[a = c]\tau$ are not open bisimilar, so there should be a formula distinguishing these processes. Such a formula is $\langle [\tau]a \parallel x \parallel \tau \parallel \tau \rangle S$, for which $\tau \models \langle [\tau]a \parallel x \parallel \tau \parallel \tau \rangle S$ and $[a = c]\tau \not\models \langle [\tau]a \parallel x \parallel \tau \parallel \tau \rangle S$. This particular construction has a bias towards $\tau$. In the classical setting of modal logic for late bisimilarity, given such a distinguishing formula, we can dualise it to obtain another distinguishing formula $\neg([\tau]a \parallel x \parallel \tau \parallel \tau)$ that has a bias towards $[a = c]\tau$, i.e., $[a = c]\tau \models \neg([\tau]a \parallel x \parallel \tau \parallel \tau) \not\models \tau \not\models \neg([\tau]a \parallel x \parallel \tau \parallel \tau)$. This dual construction fails in the case of our intuitionistic modal logic characterising open bisimilarity.

In the intuitionistic setting, we have both $\tau \not\models \neg([\tau]a \parallel x \parallel \tau \parallel \tau)$ and $[a = c]\tau \not\models \neg([\tau]a \parallel x \parallel \tau \parallel \tau)$. To address this problem, our algorithm (c.f. Section 3) simultaneously constructs two distinguishing formulae, that are not necessarily dual to each other.
The precise semantics is presented in the body of this paper. The techniques are clean and modular, so results extend to open bisimilarity for more expressive process calculi.

Outline

Section 2 introduces the semantics of Open Milner–Parrow–Walker logic (OM) and states the soundness and completeness results. Section 3 presents the proof of the correctness of an algorithm for generating distinguishing formulae, which is used to establish completeness of the logic with respect to open bisimilarity.

2 Open Milner–Parrow–Walker logic (OM)

We recall the syntax and labelled transition semantics for the finite \( \pi \)-calculus (Fig. 1). All features are standard: the deadlocked process that can do nothing, the \( \nu \) quantifier that binds private names, the output prefix that outputs a name on a channel, the input prefix that binds the name received on a channel, the silent progress action \( \tau \), the name match guard, parallel composition and non-deterministic choice. There are four types of action ranged over by \( \pi \), where a free output sends a free name, whereas a bound output extrudes a \( \nu \)-bounded private name. Stylistically, the semantics is the late labelled transition system for the \( \pi \)-calculus, where the name on the input channel is a symbolic place holder for a name that is not chosen until after an input transition.

Histories are used to define both the intuitionistic modal logic and open bisimilarity. Histories represent what is known about free variables due to how they have been communicated previously to the environment. There are two types of event to record in a history: The output of a fresh private name, using action \( a(x) \), which is denoted \( x_0 \); and a (symbolic) input, using action \( a(z) \), which is denoted \( z_0 \). The only thing that matters about the order of events in the history is the alternation between the bound outputs and symbolic inputs, since an input variable can only be instantiated with private names that were output earlier in the history. E.g., for history \( x_0 \cdot z_0 \), input variable \( z \) may be instantiated with private name \( x \); in contrast, for history \( z_0 \cdot x_0 \), input variable \( z \) may not be instantiated with private name \( x \). This is reflected by the constraints on substitutions in the following inductive definition.
A Characterisation of Open Bisimilarity using an Intuitionistic Modal Logic

Definition 1 (σ respecting h). A substitution σ invariant on names not in \( \text{fv}(h) \) is respecting a history h according to the following inductive definition.

<table>
<thead>
<tr>
<th>σ respecting ε</th>
<th>σ respecting h</th>
<th>( x \notin \text{dom}(\sigma) \cup \text{fv}(h \sigma) )</th>
<th>σ respecting ( h \cdot x' )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \sigma = \varepsilon )</td>
<td>( \sigma = h \cdot \varepsilon )</td>
<td>( x = x' )</td>
<td>( \sigma = h \cdot x'' )</td>
</tr>
</tbody>
</table>

Note that the above inductive definition fulfills the role of sets of inequality constraints called distinctions in the original work on open bisimilarity [13]. The definition above also captures the alternations between nominal and universal quantifiers in embeddings of open bisimilarity in the intuitionistic logic LINC [16, 3]. Although distinctions are more general than histories, it is shown in [16] that given a history h and its corresponding distinction D, the corresponding definitions of open bisimilarity coincide.

2.1 The semantics of the intuitionistic modal logic \( \text{OM} \)

The semantics of the modal logic \( \text{OM} \) is defined in terms of the late labelled transitions system (Fig. 2) and history respecting substitutions (Definition 1). Intuitively, each judgement must hold for all possible respectful substitutions, which explains the asymmetry between the box and diamond modalities. For the diamond modality \( \langle \pi \rangle \), a \( \pi \) transition must be possible regardless of the substitution. It is sufficient to consider the identity substitution because applying a respectful substitution cannot prevent a transition. For the box modality \( [\pi] \) there may exist substitutions \( \sigma \) other than the identity substitution enabling a \( \pi \sigma \) transition, hence we should consider all respectful substitutions.

Definition 2 (satisfaction). Process \( P \) satisfies formula \( \phi \) with history \( h \), written \( P \models^h \phi \), according to the inductive definition in Fig. 2. Satisfaction, written \( P \models \phi \), is satisfaction with a history of inputs \( x_0 \cdot \ldots \cdot x_n \), where \( \text{fv}(P) \subseteq \{x_0, \ldots, x_n\} \).

2.1.1 Why an intuitionistic modal logic?

In the open bisimulation game, every transition step is closed under respectful substitutions. Modal logic \( \text{OM} \) reflects in its semantics the substitutions that can be applied to a process. A natural semantics would be a Kripke-like semantics, where worlds are process-history pairs and the accessibility relation relates instances of such world. More precisely, consider a relation on worlds as follows: \( (P, h) \leq (Q, h') \) iff there exists a substitution \( \sigma \) respecting \( h \) such that \( P \sigma = Q \) and \( h \sigma = h' \). The pair \( (\mathcal{P}, \leq) \), where \( \mathcal{P} \) is the set of worlds, forms a Kripke frame that is reflexive and transitive. Consequently, we obtain a semantics for an intuitionistic logic, where implication is closed under respectful substitutions as follows.

\[ P \models^h \phi_1 \supset \phi_2 \quad \text{iff} \quad \forall \sigma \text{ respecting } h, P \sigma \models^h \phi_1 \sigma \implies P \sigma \models^h \phi_2 \sigma \]

Intuitionistic negation \( \neg \phi \) can then be defined as \( \phi \supset \perp \).

Recall the example from the introduction \( \overline{\pi} \models c(x) \not\models \langle \tau \rangle_{\mathbb{T}} \lor \neg \langle \tau \rangle_{\mathbb{T}} \), demonstrating that the law of excluded middle is invalid. Neither \( \overline{\pi} \models c(x) \not\models \langle \tau \rangle_{\mathbb{T}} \) nor \( \overline{\pi} \models c(x) \models \neg \langle \tau \rangle_{\mathbb{T}} \) hold. The former holds only if \( \overline{\pi} \models c(x) \) is guaranteed to make a \( \tau \) transition; but such a transition is only possible assuming \( a = c \), hence \( \overline{\pi} \models c(x) \not\models \langle \tau \rangle_{\mathbb{T}} \). For the latter, we should consider all substitutions which enable a \( \tau \) transition; and, since such a substitution \( \{\gamma \} \) exists, \( \overline{\pi} \models c(x) \not\models \neg \langle \tau \rangle_{\mathbb{T}} \). Notice that the satisfaction would hold by forcing the assumption \( a \neq c \). Of course, for open bisimilarity we make no a priori assumption about whether \( a = c \) or \( a \neq c \), since both are variables that may, or may not, be instantiated with the same value.
The intuitionistic implication and negation above are used only to explain the origin of $\mathcal{OM}$ and for contrast with properties expected of classical modal logics. The distinguishing formula algorithm, considered in subsequent sections, does not depend on these connectives.

### 2.2 Open bisimilarity, soundness and completeness

We recall the definition of open bisimilarity. Open bisimilarity is a greatest fixed point of symmetric relations closed under all respectful substitutions and labelled transitions actions at every step. Notice that a symbolic input or output of a fresh private name updates the history.

#### Definition 3 (open bisimilarity). Open bisimilarity with history $h$ is the greatest symmetric relation such that: if $P \sim^h Q$ then, for all substitutions $\sigma$ respecting $h$, the following hold, where $\alpha$ is $t$ or $\overline{t}$ and $z$ is fresh for $P$, $h$, and $\sigma$:

- $P \models^h \top$ always holds.
- $P \models^h \phi_1 \land \phi_2$ iff $P \models^h \phi_1$ and $P \models^h \phi_2$.
- $P \models^h \phi_1 \lor \phi_2$ iff $P \models^h \phi_1$ or $P \models^h \phi_2$.
- $P \models^h (x = x)\phi$ iff $P \models^h \phi$.
- $P \models^h (\alpha)\phi$ iff $\exists Q, P \xrightarrow{\pi(z)} Q$ and $Q \models^h \phi$.
- $P \models^h (\pi(z))\phi$ iff $\exists Q, P \xrightarrow{\pi(z)} Q$ and $Q \models^h z^\alpha \phi$.
- $P \models^h x = y \phi$ iff $\forall \sigma$ respecting $h$, $x\sigma = y\sigma \implies P\sigma \models^{h\sigma} \phi\sigma$.
- $P \models^h [\alpha]\phi$ iff $\forall \sigma$ respecting $h, \forall Q, P\sigma \xrightarrow{\alpha(z)} Q \implies Q \models^{h\sigma} \phi\sigma$.

$\begin{align*}
\phi &\equiv \top & \text{(true)} \\
&\equiv \bot & \text{(false)} \\
&\equiv \phi \land \phi & \text{(and)} \\
&\equiv \phi \lor \phi & \text{(or)} \\
&\equiv (x = y)\phi & \text{(dia-match)} \\
&\equiv (\pi)\phi & \text{(dia-action)} \\
&\equiv [x = y]\phi & \text{(box-match)} \\
&\equiv [\pi]\phi & \text{(box-action)} \\
\end{align*}
$

Figure 2 Syntax and semantics of the modal logic $\mathcal{OM}$, where $\alpha$ is $t$ or $\overline{t}$; and $z$ is fresh for $P$, $h$, and $\sigma$. 

Open bisimilarity, written $P \sim Q$, is defined to be open bisimilarity with a history $x_0^i, \ldots, x_n^i$ such that $\text{fv}(P) \cup \text{fv}(Q) \subseteq \{x_0, \ldots, x_n\}$. 

\[ P \models^h \tau \]  
\[ P \models^h \phi_1 \land \phi_2 \text{ iff } P \models^h \phi_1 \text{ and } P \models^h \phi_2. \]  
\[ P \models^h \phi_1 \lor \phi_2 \text{ iff } P \models^h \phi_1 \text{ or } P \models^h \phi_2. \]  
\[ P \models^h (x = x)\phi \text{ iff } P \models^h \phi. \]  
\[ P \models^h (\alpha)\phi \text{ iff } \exists Q, P \xrightarrow{\pi} Q \text{ and } Q \models^h \phi. \]  
\[ P \models^h (\pi(z))\phi \text{ iff } \exists Q, P \xrightarrow{\pi(z)} Q \text{ and } Q \models^h z^\alpha \phi. \]  
\[ P \models^h x = y \phi \text{ iff } \forall \sigma \text{ respecting } h, x\sigma = y\sigma \implies P\sigma \models^{h\sigma} \phi\sigma. \]  
\[ P \models^h [\alpha]\phi \text{ iff } \forall \sigma \text{ respecting } h, \forall Q, P\sigma \xrightarrow{\alpha(z)} Q \implies Q \models^{h\sigma} \phi\sigma. \]  
\[ h \equiv \epsilon \text{ (empty)} \]  
\[ h \cdot x^\alpha \text{ (name)} \]  
\[ h \cdot x^i \text{ (variable)} \]  
\[ \phi \equiv \top \text{ (true)} \]  
\[ \phi \equiv \bot \text{ (false)} \]  
\[ \phi \equiv \phi \lor \phi \text{ (or)} \]  
\[ \phi \equiv \phi \land \phi \text{ (and)} \]  
\[ \phi \equiv (x = y)\phi \text{ (dia-match)} \]  
\[ \phi \equiv (\pi)\phi \text{ (dia-action)} \]  
\[ \phi \equiv [x = y]\phi \text{ (box-match)} \]  
\[ \phi \equiv [\pi]\phi \text{ (box-action)} \]
2.2.1 Soundness and completeness results

The main result of this paper is that, for finite π-calculus processes open bisimilarity (∼) coincides with the relation between processes with no distinguishing formula (⇝).

► **Definition 4** (logical equivalence). $P \leftrightarrow Q$ is defined whenever, for all φ, $P \models \phi$ iff $Q \models \phi$.

► **Theorem 5** (soundness). For π-calculus processes (including replication), $P \sim Q$ implies $P \leftrightarrow Q$.

► **Theorem 6** (completeness). For finite π-calculus processes, $P \leftrightarrow Q$ implies $P \sim Q$.

The proof of soundness has been mechanically checked in the proof assistant Abella [2] using the two-level logic approach [4] to reason about the π-calculus semantics specified in λProlog [11]. The proof of soundness proceeds by induction on the structure of the logical formulae in the definition of logical equivalence. The proof of completeness is explained in detail in Section 3. Soundness extends to infinite π-calculus processes with replication, but completeness holds for decidable fragments such as Fig. 1.

Firstly, we provide examples demonstrating the implications of Theorems 5 and 6. Due to soundness, if two processes are bisimilar, we cannot find a distinguishing formula that holds for one process but does not hold for the other process. Due to completeness, if it is impossible to prove that two process are open bisimilar, then we can construct a distinguishing formula that holds for one process but does not hold for the other process. Thus Theorems 5 and 6 guarantee that an OOM formulae can be used to characterise non-bisimilarity.

2.2.2 Example processes distinguishable by postconditions

All modalities are essential for the soundness and completeness of OOM. Perhaps the least obvious modality is $\langle x = y \rangle$. When prefixed with a box modality it indicates a postcondition that always holds after an action. To see, this consider the process $[x = y]τ$. The judgement $[x = y]τ \models \langle τ \rangle \langle x = y \rangle τ$ holds since for any θ such that $([x = y]τ)θ \Downarrow 0$ it must be the case that $xθ = yθ$. Hence, by definition of diamond, $0 \models \langle xθ = yθ \rangle τ$ iff $0 \models τ$. In contrast, $τ \not\models \langle τ \rangle \langle x = y \rangle τ$ since, taking the identity substitution in the definition of box, $τ \Downarrow 0$, but $0 \models \langle x = y \rangle τ$ cannot be proven in general. The formula $\langle τ \rangle \langle x = y \rangle τ$ is therefore a distinguishing formula satisfied by $[x = y]τ$ but not $τ$.

The use of $\langle x = y \rangle$ as a postcondition contrasts to the use of $[x = y]$ as a precondition. Consider the same process as above with the formula $[x = y]⟨ τ ⟩ τ$. Observe that, for substitutions θ such that $xθ = yθ$, $([x = y]τ)θ \Downarrow 0$ and $0 \models τ$ holds, hence $([x = y]τ)θ \models ⟨ τ ⟩ τ$; and thereby the judgement $[x = y]τ \models [x = y]⟨ τ ⟩ τ$ holds. In contrast, $0 \not\models [x = y]⟨ τ ⟩ τ$.

We will return to the above two formulae shortly, as they are critical for the algorithm for generating distinguishing formulae.

2.3 Sketch of algorithm for generating distinguishing formulae

The completeness proof in Section 3 relies on an algorithm for generating distinguishing formulae for non-bisimilar processes. Here, we provide a sketch of the algorithm executed on key examples.

2.3.1 Example requiring intuitionistic assumptions

The algorithm proceeds over the structure of a tree of moves that show two processes are non-bisimilar. In base cases, we have a pair of processes where, under a substitution, one
A process can make a transition, but the other process cannot match the transition. We revisit two examples of base cases, discussed previously:

\[ x = y \] \( \tau \not\sim 0 \): The left process leads by \( \{ x = y \} \{ \tau \} \downarrow 0 \), but 0 cannot make a \( \tau \) transition, under any substitution; hence \( [x = y] \tau \models [x = y] (\tau) \) and 0 \( \not\models [\tau] \). are distinguishing formulae.

\[ x = y \] \( \tau \not\sim \tau \): The right process leads by \( \tau \downarrow 0 \), but \( [x = y] \tau \theta \) can make a \( \tau \) transition only when \( x \theta = y \theta \); hence \( [x = y] \tau \models [\tau] (x = y) \) and \( \tau \models (\tau) \) are distinguishing formulae.

In an inductive case, the two processes cannot be distinguished by an immediate transition. However, under some substitutions, one process can make a \( \tau \) transition to a state, say \( P' \), that, under the same substitution the other process can only make a corresponding \( \pi \) transition to reach states \( Q_i \) that are non-bisimilar to \( P' \). This allows a distinguishing formula to be inductively constructed from the distinguishing formulae for \( P' \) paired with each \( Q_i \).

For example, consider how the algorithm would find distinguishing formulae for \( P \) and \( Q \) below.

\[
\begin{align*}
P &\triangleq \pi.([x = y] \tau + \tau + \tau. \tau) \\
P' &\triangleq [x = y] \tau \\
0 &\models Q_1 \\
\tau &\models Q_2
\end{align*}
\]

The first step in the strategy for non-bisimilarity is to show that \( P \) can make a \( \tau \) transition to a state that is not bisimilar to any state reachable by a \( \tau \) transition from the other process. One possibility is the transition to \( P' \) as illustrated above. In reply, \( Q \) may attempt a corresponding \( \tau \) transition either to \( Q_1 \) or \( Q_2 \). Inductively, we require that \( P' \not\sim Q_1 \) and \( P' \not\sim Q_2 \). Both are instances of the base case discussed above where we discovered distinguishing formulae for each of them.

This enables us to construct distinguishing formulae for the inductive case. The distinguishing formula satisfied by \( P \) is a diamond followed by the conjunction of the left distinguishing formulae that is satisfied by \( P' \) in the base cases: \( \pi.([x = y] \tau + \tau + \tau. \tau) \models (\tau)[(x = y) \tau \land [x = y] (\tau) \) \). The distinguishing formula satisfied by \( Q \) is a box followed by the disjunction of the right distinguishing formulae from the base cases: \( \tau + \tau. \tau \models (\tau) (\tau) \lor (\tau) \).

To confirm that they are indeed distinguishing formulae for \( P \) and \( Q \), swap the processes and formulae above to observe that each process fails to satisfy the other formula. To be precise, assume for contradiction that \( \tau + \tau. \tau \models (\tau)([x = y] \tau \land [x = y] (\tau) \) holds. By definition of \( (\tau) \), this holds iff either \( 0 \models [x = y] (\tau) \tau \land [x = y] (\tau) \) or \( \tau \models [x = y] (\tau) \tau \land [x = y] (\tau) \) holds. Now observe that \( 0 \models [x = y] (\tau) \tau \) holds iff we make the additional assumption in the meta framework that \( x \) and \( y \) are persistently distinct, i.e., for all \( \sigma, x \sigma \neq y \). In addition, observe that \( \tau \models [x = y] (\tau) \tau \) holds iff we make the additional assumption in the meta framework that \( x \) and \( y \) are persistently equal, i.e., for all \( \sigma, x \sigma = y \). In fact, by these observations we are able to mechanically prove the following in intuitionistic framework Abella: \( \tau + \tau. \tau \models (\tau)([x = y] \tau \land [x = y] (\tau) \) iff \( \forall x, y, (x = y \lor x \neq y) \). Notice that \( \forall x, y, (x = y \lor x \neq y) \) is an instance of the law of excluded middle; hence, assuming the law of excluded middle, the formula for \( Q \) also holds for \( P \); and vice versa. Indeed there would be no distinguishing formulae for these processes; and hence in a classical framework the modal logic would be incomplete. Fortunately, since intuitionistic logics do not assume the law of excluded middle, as long as we evaluate the semantics in an intuitionistic framework, we are able to establish that \( Q \not\models (\tau)([x = y] \tau \land [x = y] (\tau) \), as required.
2.3.2 Example involving private names that are distinguishable

The alternation between inputs and outputs in the history affects what counts as a respectful substitution. Intuitively, respectful substitutions ensure that a private name can never be input earlier than it was output. Consider the following processes: $P \triangleq \nu x. a(x). \tau \sim \nu x. a(x). [x = y] \tau \triangleq Q$.

These processes are not open bisimilar because $P$ can make the following three transition steps: $\nu x. a(x). \tau \xrightarrow{a(x)} a(y). \tau \xrightarrow{a(y)} \tau \xrightarrow{0} 0$. However, $Q$ can only match the first two steps. At the third step, a base case of the algorithm for non-bisimilarity is reached where $[x = y] \tau$ applies. In this case, any substitution $\theta$ respecting $a^x a^y$ such that $[x = y] \tau \theta \xrightarrow{0} 0$ is such that $\theta \gamma = x$, $x \not\in \text{dom}(\theta)$ and $a\theta \neq x$, which is satisfiable. Thus $[x = y] \tau \models \langle x = y \rangle \Box$ and $\tau \models \langle x = y \rangle \Box$. By applying inductive cases, we obtain $\nu x. a(x). \tau \models \langle \Pi(x) \rangle \langle a(y) \rangle \langle \tau \rangle \Box$ and $\nu x. a(x). [x = y] \tau \models \langle a(y) \rangle \langle \tau \rangle \Box$.

2.3.3 Example involving private names that are indistinguishable

In contrast to the previous example, consider the following processes where a fresh name is output and compared to a name already known: $\nu x. \Pi x \sim \nu x. \Pi x. [x = a] \tau$.

The existence of a distinguishing formula of the form $\langle \Pi(x) \rangle [x = a] \langle \tau \rangle \Box$ is prevented by the history. Both $\nu x. \Pi x. [x = a] \tau \models \langle \Pi(x) \rangle [x = a] \langle \tau \rangle \Box$ and $\nu x. \Pi x \models \langle \Pi(x) \rangle [x = a] \langle \tau \rangle \Box$ hold. The latter holds since $\nu x. \Pi x \models \Pi^x [x = a] \langle \tau \rangle \Box$ holds if and only if $\nu x. \Pi x \models \Pi^x [x = a] \langle \tau \rangle \Box$ holds.

By definition of $[x = a]$, this holds if only if for all $\theta$ respecting $a^x a^y$ and such that $x\theta = a\theta$, $0 \models \langle a^x a^y \rangle \langle \tau \rangle \Box$. Clearly $0$ cannot make a $\tau$ transition, hence $0 \models \langle a^x a^y \rangle \langle \tau \rangle \Box$ does not hold. However, fortunately, there is no substitution $\theta$ respecting $a^x a^y$ such that $x\theta = a\theta$. By the definition of respecting substitution, $\theta$ must satisfy $x \not\in \text{dom}(\theta)$ and $x \neq a\theta$, contradicting constraint $x\theta = a\theta$. Thereby $0 \models \langle a^x a^y \rangle [x = a] \langle \tau \rangle \Box$ holds vacuously; hence $\nu x. \Pi x \models \Pi^x [x = a] \langle \tau \rangle \Box$ holds as required.

3 Completeness of open bisimilarity with respect to OM

There is a constructive definition of non-bisimilarity. Since bisimilarity is defined in terms of a greatest fixed point of relations satisfying a certain closure property, non-bisimilarity is defined in terms of a least fixed point satisfying the dual property. This leads to the following constructive definition of non-bisimilarity from which a non-bisimilarity algorithm can be extracted. Since non-bisimilarity is defined in terms of a least fixed point, there is a finite winning strategy, consisting of a finite tree of moves such that in each branch eventually a pair of processes and a history is reached such that one process can make a move that the other cannot always match.

**Definition 7** (non-bisimilarity). Firstly, we inductively define the family of relation $\not\sim_n^h$, for $n \in \mathbb{N}$. The base case is when, for some respectful substitution one player can make a move, that cannot be matched by the other player without assuming a stronger substitution.

The class of all such pairs of processes form the base case for the construction of the non-bisimilarity relation, say $P \not\sim^h Q$. More precisely, the relation $\not\sim_n^h$ is the least symmetric relation such that for any $P$ and $Q$, $P \not\sim_n^h Q$ whenever there exist process $P'$, action $\pi$ and substitution $\sigma$ respecting $h$, such that the following holds:

$P \sigma \xrightarrow{\pi} P'\text{, for } x \in \text{bn}(\pi), x \text{ is fresh for } P \sigma, Q \sigma \text{ and } h \sigma, \text{ and there is no } Q'\text{ such that } Q \sigma \xrightarrow{\pi} Q'$. 

Inductively, \( \not\sim^h_{n+1} \) is the least symmetric relation extending \( \not\sim^h_n \) such that \( P \not\sim^h_{n+1} Q \) whenever for some substitution \( \sigma \) respecting \( h \), one of the following holds, where \( \alpha \) is \( \tau \) or \( \pi \): 
\[
\begin{align*}
P\sigma &\xrightarrow{\tau} P' \text{ and for all } Q_i, \text{ such that } Q\sigma \xrightarrow{\tau} Q_i, \text{ } P' \not\sim^h_n Q_i. \\
P\sigma &\xrightarrow{\pi(x)} P', \text{ and for all } Q_i \text{ and } x \text{ fresh for } P\sigma, Q\sigma \text{ and } h\sigma, \text{ such that } Q\sigma \xrightarrow{\pi(x)} Q_i, \\
P' &\not\sim^h_n x^\sigma Q_i. \\
P\sigma &\xrightarrow{\pi(x)} P', \text{ and for all } Q_i \text{ and } x \text{ fresh for } P\sigma, Q\sigma \text{ and } h\sigma, \text{ such that } Q\sigma \xrightarrow{\pi(x)} Q_i, \\
P' &\not\sim^h_n x^\sigma Q_i.
\end{align*}
\]

Thereby, the relation \( P \not\sim^h_n Q \) contains all processes that can be distinguished by a strategy with depth at most \( n \), i.e., at most \( n \) moves are required to reach a pair of processes in \( \not\sim^h_0 \), at which point there is an accessible world in which a process can make a move that the other process cannot match.

The relation \( \not\sim^h \), pronounced non-bisimilarity with history \( h \), is defined to be the least relation containing \( \not\sim^h_n \) for all \( n \in \mathbb{N} \), i.e. \( \bigcup_{n \in \mathbb{N}} \not\sim^h_n \). Similarly to open bisimulation, \( P \not\sim Q \) is defined as \( P \not\sim^x_1 \ldots^x_m Q \) where \( \text{fv}(P) \cup \text{fv}(Q) \subseteq \{x_1, \ldots, x_m\} \).

### 3.1 Preliminaries

We require the following terminology for substitutions, and abbreviations for formulae.

- **Definition 8.** Composition of substitutions \( \sigma \) and \( \theta \) is defined such that \( P[\sigma \cdot \theta] = (P\sigma)\theta \), for all processes \( P \). For substitutions \( \sigma \) and \( \theta \), \( \sigma \preceq \theta \) whenever there exists \( \sigma' \) such that \( \sigma \cdot \sigma' = \theta \). For a finite substitution \( \sigma = \{z_1/x_1\} \ldots \{z_n/x_n\} \) the formula \( [\sigma]\phi \) abbreviates the formula \( [x_n = z_n] \ldots [x_1 = z_1] \phi \). Similarly, \( \langle \sigma \rangle \phi \) abbreviates \( \langle x_n = z_n \rangle \ldots \langle x_1 = z_1 \rangle \phi \). For finite set of formulae \( \phi_i \), formula \( \bigvee_i \phi_i \) abbreviates \( \phi_1 \lor \ldots \lor \phi_n \), where the empty disjunction is \( \mathbf{t} \). Similarly \( \bigwedge_i \phi_i \) abbreviates \( \phi_1 \land \ldots \land \phi_n \), where the empty conjunction is \( \mathbf{f} \).

We require the following technical lemmas. The first (image finiteness, as used in [5]) ensures that there are finitely many reachable states in one step, up to renaming. The second extends the definition of the box-match modality to finite substitutions. The third is required in inductive cases involving bound output and input. The fourth is a monotonicity property for satisfaction. The fifth is a monotonicity property for transitions ensuring names bound by label are not changed by a substitution.

- **Lemma 9.** For process \( P \) and action \( \pi \) there are finitely many \( P_i \) such that \( P \xrightarrow{\pi} P_i \).

- **Lemma 10.** If for all \( \theta \) respecting \( h \) and \( \sigma \preceq \theta \), it holds that \( P\theta \models^{h\sigma} \phi \theta \), then \( P \models^h [\sigma] \phi \) holds.

- **Lemma 11.** If \( \sigma \cdot \theta \) respects \( h \), then \( \theta \) respects \( h\sigma \).

- **Lemma 12.** If \( P \models^h \phi \) holds then \( P\theta \models^{h\sigma}\phi \theta \) holds for any \( \theta \) respecting \( h \).

- **Lemma 13.** If \( P \xrightarrow{\gamma} Q \) then \( P\theta \xrightarrow{\gamma\theta} Q\theta \), for all \( \theta \) such that if \( x \in \text{bn}(\gamma) \) and \( y\theta = x \) then \( x = y \).

### 3.2 Algorithm for distinguishing formulae

The constructive definition of non-bisimilarity gives a tree of substitutions and actions forming a strategy showing that two processes are not open bisimilar. The following proposition shows that CM formulae are sufficient to capture such strategies. For any strategy that distinguishes two processes, we can construct distinguishing CM formulae. A distinguishing
A Characterisation of Open Bisimilarity using an Intuisionistic Modal Logic

formula holds for one process but not for the other process. Furthermore, there are always
at least two distinguishing formulae, one biased to the left and another biased to the right,
as in the construction of the proof for the following proposition. As discussed in the intro-
duction, the left bias cannot be simply obtained by negating the right bias and vice versa;
both must be constructed simultaneously and may be unrelated by negation.

Proposition 14. If \( P \not\sim Q \) then there exists \( \phi_L \) such that \( P \models \phi_L \) and \( Q \not\models \phi_L \), and also
there exists \( \phi_R \) such that \( Q \models \phi_R \) and \( P \not\models \phi_R \).

Proof. Since \( \not\sim^h \) is defined by a least fixed point over a family of relations \( \not\sim^h_n \), if \( P \not\sim^h Q \),
there exists \( n \) such that \( P \not\sim^h_n Q \), so we can proceed by induction on the depth of a winning
strategy.

In the base case, assume that \( P \not\sim^h_0 Q \), hence by definition, for substitution \( \sigma \) respecting
\( h, P\sigma \xrightarrow{\sigma} P' \), for \( x \in \text{bn}(\pi) \), \( x \) is fresh for \( P\sigma, Q\sigma \) and \( h\sigma \), such that there is no \( Q' \) such that
\( Q\sigma \xrightarrow{\sigma} Q' \), up to symmetry of \( \not\sim^h_n \). There exist finitely many pairs of variables \( x_j \)
and \( y_j \), selected from \( \text{fv}(P) \cup \text{fv}(Q) \cup \text{fv}(\pi) \) such that \( x_j\sigma \neq y_j\sigma \), and, for any \( R \) and substitution \( \theta \)
respecting \( h \), if \( Q\theta \xrightarrow{\theta} R \) there exists \( j \) such that \( x_j\theta = y_j\theta \). To see why, assume for
contradiction that there is some \( \theta \) respecting \( h \) such that \( Q\theta \xrightarrow{\theta} R \) but there is no \( x \) and \( y \)
in \( \text{fv}(P) \cup \text{fv}(Q) \cup \text{fv}(\pi) \) such that \( x\sigma \neq y\sigma \) and \( x\theta = y\theta \). Stated otherwise, for all \( x \) and \( y \)
in \( \text{fv}(P) \cup \text{fv}(Q) \cup \text{fv}(\pi) \) if \( x\theta = y\theta \) then \( x\sigma = y\sigma \), which is precisely the definition of a
function, i.e., substitution, say \( \theta' \), defined on \( \text{fv}(P\theta) \cup \text{fv}(Q\theta) \cup \text{fv}(\pi\theta) \) such that \( \theta' \) maps \( z\theta \)
to \( z\sigma \). In that case, \( \theta \circ \theta' = \sigma \) on \( \text{fv}(P) \cup \text{fv}(Q) \cup \text{fv}(\pi) \); and hence, by Lemma 13, since for
\( x \in \text{bn}(\pi) \), \( x \) is fresh, \( Q\theta\circ\theta' \not\sim^h \) \( R\theta' \) contradicting the initial assumption for the base case that
no transition \( Q\sigma \xrightarrow{\sigma} Q' \) exists for any \( Q' \).

In this case, there are two distinguishing formulae \([\sigma](\pi)\texttt{t} \) and \([\pi]\bigvee_j(x_j=y_j)\texttt{t} \) biased
to \( P \) and \( Q \) respectively. There are four cases to check to confirm that these are distinguishing
formulae.

Case \( P \models^h [\sigma](\pi)\texttt{t} \): Consider all \( \theta \) respecting \( h \) such that \( \sigma \leq \theta \). By definition there
exists \( \theta' \) such that \( \sigma \circ \theta' = \theta \), so since \( P\sigma \xrightarrow{\sigma} P' \), by Lemma 13, \( P\theta \xrightarrow{\theta} P' \theta' \). Thereby,
since \( P'\theta' \models^h \texttt{t} \), holds, \( P\theta \models^h (\pi\theta)\texttt{t} \). Hence, by Lemma 10, \( P \models^h [\sigma](\pi)\texttt{t} \).

Case \( Q \not\models^h [\sigma](\pi)\texttt{t} \): Assume \( Q \models^h [\sigma](\pi)\texttt{t} \) for contradiction. Now, since \( \sigma \) respects \( h \)
and \( \sigma \leq \theta \), by Lemma 10, \( Q \models^h [\sigma](\pi)\texttt{t} \) holds only if \( Q\sigma \models^h (\pi\sigma)\texttt{t} \) holds; which
holds only if there exists \( Q' \) such that \( Q\sigma \xrightarrow{\sigma} Q' \), contradicting the assumption that no
such \( Q' \) exists. Thereby \( Q \not\models^h [\sigma](\pi)\texttt{t} \).

Case \( Q \models^h [\pi]\bigvee_j(x_j=y_j)\texttt{t} \): Consider substitutions \( \theta \) respecting \( h \) and \( Q' \) such that
\( Q\theta \xrightarrow{\theta} Q' \). It must be the case that there exists \( j \) such that \( x_j\theta = y_j\theta \), thereby
\( Q' \models^h \langle x_j\theta = y_j\theta \rangle\texttt{t} \) holds; hence clearly \( Q' \models^h \left( \bigvee_j \langle x_j=y_j \rangle\texttt{t} \right) \theta \) holds. Hence
\( Q \models^h [\pi]\bigvee_j(x_j=y_j)\texttt{t} \).

Case \( P \not\models^h [\pi]\bigvee_j(x_j=y_j)\texttt{t} \): Assume for contradiction \( P \models^h [\pi]\bigvee_j(x_j=y_j)\texttt{t} \). This
holds iff for all processes \( S \) and substitutions \( \theta \) respecting \( h \), \( P\theta \xrightarrow{\theta} S \) implies \( S \models^{h'} \)
\( \left( \bigvee_j \langle x_j=y_j \rangle\texttt{t} \right) \theta \). Since we know that \( \sigma \) respects \( h \) and \( P\sigma \xrightarrow{\sigma} P' \), we have \( P' \models^{h''} \)
\( \left( \bigvee_j \langle x_j=y_j \rangle\texttt{t} \right) \sigma \). This holds only if for some \( j \), \( P' \models^{h''} \langle x_j\sigma = y_j\sigma \rangle\texttt{t} \); hence, \( x_j\sigma = y_j\sigma \) for some \( j \), which
contradicts the assumption that \( x_j\sigma \neq y_j\sigma \). Thereby \( P \not\models^h [\pi]\bigvee_j(x_j=y_j)\texttt{t} \).

Now consider the inductive cases. Given \( P, Q \), if \( P \not\sim^h_{n+1} Q \), up to symmetry of \( \not\sim^h_{n+1} \),
there are three cases to consider, for some substitution \( \sigma \) respecting \( h \), where \( \alpha \) is either \( \tau \)
or \( \overline{\alpha}b \):
\( P_\sigma \xrightarrow{a_\sigma} P' \) and for all \( Q \), such that \( Q_\sigma \xrightarrow{a_\sigma} Q_i, P'_\sigma \not\xrightarrow{a_\sigma} Q_i \).

\( P_\sigma \xrightarrow{a_\sigma(x)} P', \) and for all \( Q_i \), and \( x \) fresh for \( P_\sigma, Q_\sigma \) and \( h_\sigma \), such that \( Q_\sigma \xrightarrow{a_\sigma(x)} Q_i, P'_\sigma \not\xrightarrow{a_\sigma(x)} Q_i \).

\( P_\sigma \xrightarrow{a_\sigma(x)} P', \) and for all \( Q_i \), and \( x \) fresh for \( P_\sigma, Q_\sigma \) and \( h_\sigma \), such that \( Q_\sigma \xrightarrow{a_\sigma(x)} Q_i, P'_\sigma \not\xrightarrow{a_\sigma(x)} Q_i \).

We consider the second case above involving bound output only, the other two cases are similar — differing only in the accounting for respectful substitutions according to Def. 1.

For \( P_\sigma \xrightarrow{a_\sigma(x)} P' \), by Lemma 9, there exist finitely many \( Q_i \) such that \( Q_\sigma \xrightarrow{a_\sigma(x)} Q_i \). For each \( i \), since \( P'_\sigma \not\xrightarrow{a_\sigma(x)} Q_i \), by the induction hypothesis, there exist \( \phi^L_i \) and \( \phi^R_i \) such that \( P'_\sigma \models h_\sigma \cdot x^o \phi^L_i \sigma \) and \( Q_i \not\models h_\sigma \cdot x^o \phi^L_i \sigma \) and \( P'_\sigma \not\models h_\sigma \cdot x^o \phi^R_i \sigma \) and \( Q_i \models h_\sigma \cdot x^o \phi^R_i \sigma \). Furthermore, assume that \( \sigma \) is minimal with respect to the order over substitutions in the sense that, if \( \theta \leq \sigma \) is such that \( P_\sigma \xrightarrow{a_\sigma(x)} P'' \), \( P'' \models h_\sigma \cdot x^o \), \( \theta \) and for all \( Q'' \) such that \( Q_\sigma \xrightarrow{a_\sigma(x)} Q'' \), \( P'' \models h_\sigma \cdot x^o \), then \( \theta = \sigma \).

By a similar argument to the base case, there are finitely many pairs of variables \( x_j \) and \( y_j \) selected from \( \text{fv}(P) \cup \text{fv}(Q) \cup \{u\} \) such that \( x_j \sigma \not\equiv y_j \sigma \) and, for any substitution \( \theta \) respecting \( h \), if, for some \( S, Q_\sigma \xrightarrow{a_\sigma(x)} S \) then either: there exists some \( j \) such that \( x_j \theta = y_j \theta \); or both \( \sigma \leq \theta \) and \( \theta \leq \sigma \) hold and hence there exist \( i \) and \( \theta' \) such that \( \sigma \cdot \theta' = \theta \) and \( S, \theta' = S \). Notice cases where \( \theta < \sigma \) are eliminated by minimality of \( \sigma \).

From the above, dismissing formulae \( [\sigma](\pi(x)) \land_i \phi^L_i \) and \( [\pi(x)](\lor_j (x_j = y_j)) \) can be constructed. There are four cases to consider to verify these are indeed dismissing formulae.

**Case \( P \models [\sigma](\pi(x)) \land_i \phi^L_i :** Consider all \( \theta \) such that \( \sigma \leq \theta \), \( \theta \) respects \( h \), and without loss of generality \( x \) is fresh such that \( x \not\in \text{dom}(\theta) \) and \( x \not\in \text{fv}(h_\theta) \). By definition, there exists \( \theta' \) such that \( \sigma \cdot \theta' = \theta \). Now since \( \sigma \cdot \theta' \) respects \( h \sigma \), by Lemma 11, \( \theta' \) respects \( h_\sigma \) hence since \( x \not\in \text{dom}(\theta') \) and \( x \not\in \text{fv}(h_\theta) \), \( \theta' \) respects \( h_\sigma \cdot x^o \). Subsequently \( \theta' \) respects \( h_\sigma \cdot x^o \) and also \( P'_\sigma \models h_\sigma \cdot x^o \phi^L_i \sigma \) holds, by Lemma 12, it holds that \( P'\theta' \models h_\sigma \cdot x^o \phi^L_i \theta \). The above holds for all \( i \), hence it holds that \( P'\theta' \models h_\sigma \cdot x^o \lor_i \phi^L_i \theta \). Now, since \( P_\sigma \xrightarrow{a_\sigma(x)} P' \), by Lemma 12, since \( x \) is fresh, \( P_\sigma \xrightarrow{a_\sigma(x)} P' \theta' \) holds; and hence \( P_\sigma \models h_\theta (\langle \pi(x) \rangle \land_i \phi^L_i \theta) \) holds. Therefore, by Lemma 10, \( P \models [\sigma](\langle \pi(x) \rangle) \land_i \phi^L_i \theta \).

**Case \( Q \not\models [\sigma](\langle \pi(x) \rangle) \land_i \phi^L_i :** Assume for contradiction that \( Q \models [\sigma](\langle \pi(x) \rangle) \land_i \phi^L_i \). Since \( \sigma \) respects \( h \) and \( \sigma \leq \sigma \), by Lemma 10, the above assumption holds only if \( Q_\sigma \models h_\sigma (\langle \pi(x) \rangle) \land_i \phi^L_i \sigma \) holds. Now \( Q_\sigma \models h_\sigma (\langle \pi(x) \rangle) \land_i \phi^L_i \sigma \) holds only if there exists \( Q' \) such that \( Q_\sigma \xrightarrow{a_\sigma(x)} Q' \) and \( Q' \models h_\sigma \cdot x^o \lor_i \phi^L_i \sigma \), which holds only if \( Q' \models h_\sigma \cdot x^o \lor_i \phi^L_i \sigma \) for all \( i \). Notice that \( Q' = Q_k \) for some \( k \), and therefore \( Q_k \models h_\sigma \cdot x^o \lor_i \phi^L_i \sigma \), but it was assumed that \( Q_k \not\models h_\sigma \cdot x^o \lor_i \phi^L_i \sigma \) leading to a contradiction. Therefore \( Q \not\models [\sigma](\langle \pi(x) \rangle) \land_i \phi^L_i \).

**Case \( Q \models h_\theta (\lor_j (x_j = y_j) \pi) :** Fix \( Q' \) and \( \theta \) respecting \( h \) such that \( Q_\theta \xrightarrow{a_\sigma(x)} Q' \) and without loss of generality assume \( x \) is fresh such that \( x \not\in \text{dom}(\theta) \) and \( x \not\in \text{fv}(h_\theta) \). There are two sub-cases to consider. Firstly consider where for some \( k \), \( x_k \theta = y_k \theta \), in which case it holds that \( Q'_\sigma \models h_\theta \cdot x^o \langle x_k \theta = y_k \theta \rangle \pi \), and hence \( Q'_\sigma \models h_\theta \cdot x^o (\lor_i \phi^L_i \pi \lor_j (x_j = y_j) \pi) \theta \), by definition of disjunction. Secondly consider where there exists \( \theta' \) such that \( \sigma \cdot \theta' = \theta \) and for some \( \ell, Q_\sigma \xrightarrow{a_\sigma(x)} Q_\ell \) such that \( Q_{\ell \theta'} = Q' \). Now since \( \sigma \cdot \theta' \) respects \( h \), by Lemma 11, \( \theta' \) respects \( h_\sigma \), hence by definition of respectful substitutions, since \( x \not\in \text{dom}(\theta) \) and \( x \not\in \text{fv}(h_\theta) \), \( \theta' \) respects \( h_\sigma \cdot x^o \). Therefore, by Lemma 12, since \( Q_\ell \models h_\sigma \cdot x^o \phi^L_i \sigma \) and \( \theta' \) respects \( h_\sigma \cdot x^o \), \( Q_{\ell \theta'} \models h_\sigma \cdot x^o \phi^L_i \theta \) holds. Hence \( Q'_\sigma \models h_\sigma \cdot x^o (\lor_i \phi^L_i \pi \lor_j (x_j = y_j) \pi) \theta \), by definition of disjunction. Thus by definition of \( \langle \pi(x) \rangle \), we can conclude that \( Q \models h_\theta (\langle \pi(x) \rangle) (\lor_i \phi^L_i \pi \lor_j (x_j = y_j) \pi) \) holds.
Case \( P \not\prec^h [\pi(x)] (\bigvee_i \phi_i^R \vee \bigvee_j \langle x_j = y_j \rangle \tau) \): Assume \\
\( P \models^h [\pi(x)] (\bigvee_i \phi_i^R \vee \bigvee_j \langle x_j = y_j \rangle \tau) \) for contradiction. Since \( \sigma \) respects \( h \) and \( P \models^h \pi(x) \), \( P' \), the previous assumption can only hold if \( P' \models^{h_o} \pi(x) \) \( (\bigvee_i \phi_i^R \vee \bigvee_j \langle x_j = y_j \rangle \tau) \). This holds only if, for some \( i \), \( P' \models^{h_o} \phi_i^R \) or, for some \( j \), \( P' \models^{h_o} \langle x_j = y_j \rangle \tau \). However, for all \( i \), \( P' \not\models^{h_o} \phi_i^R \) and, also for all \( j \), we have \( x_j \not\models y_j \tau \) and \( P' \not\models^{h_o} \langle x_j = y_j \rangle \tau \), leading to a contradiction in either case. Thereby \( P \not\prec^h [\pi(x)] \) \( (\bigvee_i \phi_i^R \vee \bigvee_j \langle x_j = y_j \rangle \tau) \).

By induction we have established that, for any history \( h \), processes \( P \) and \( Q \), and any \( n \), if \( P \not\prec^n Q \) then there exists \( \phi_L \) such that \( P \models^h \phi_L \) and \( Q \not\prec^n \phi_L \), and also there exists \( \phi_R \) such that \( Q \models^h \phi_R \) and \( P \not\prec^h \phi_R \). The result then follows by observing that \( \not\prec^h \) is the least relation containing all \( \not\prec^n \); and, furthermore, \( P \not\prec Q \) holds simply when \( P \not\prec^1 \cdots x^n Q \) holds, where \( \text{fv}(P) \cup \text{fv}(Q) \subseteq \{x_1, \ldots, x_n\} \).

Since open bisimilarity is decidable for finite \( \pi \)-calculus processes, the constructive non-bisimilarity in Definition 7 coincides with the negation of open bisimilarity.

\textbf{Lemma 15.} For finite processes, \( P \not\sim Q \) holds, according to constructive non-bisimilarity in Definition 7, if and only if \( P \sim Q \) does not hold.

Combining Proposition 14 with Lemma 15 yields immediately the completeness of \( \text{OM} \) with respect to open bisimilarity. Completeness (Theorem 6) establishes that the set of all pairs of processes that have the same set of distinguishing formulae is an open bisimilarity. The proof can now be stated as follows.

\textit{Proof of Theorem 6:} Assume that for finite processes \( P \) and \( Q \), for all formulae \( \phi \), \( P \models \phi \) iff \( Q \models \phi \). Now for contradiction suppose that \( P \sim Q \) does not hold. By Lemma 15, \( P \not\prec Q \) must hold. Hence by Proposition 14 there exists \( \phi_L \) such that \( P \models \phi_L \) but \( Q \not\models \phi_L \), but by the assumption above \( Q \models \phi_L \), leading to a contradiction. Thereby \( P \sim Q \).

Notice that soundness (Theorem 5) and the non-bisimilarity algorithm (Proposition 14) also hold for infinite \( \pi \)-calculus processes (using replication for instance). However, for infinite \( \pi \)-calculus processes, open bisimilarity is undecidable; hence additional insight may be needed to justify whether Lemma 15 holds for infinite processes. Thereby in the infinite case, while it is impossible that \( P \sim Q \) and \( P \not\sim Q \) holds, it may be the case that neither holds. A possibility is that a more expressive logic is required to completely characterise open bisimilarity for infinite processes.

### 3.3 Example runs of distinguishing formulae algorithm

We provide further examples of non-bisimilar processes that illustrate subtle aspects of the algorithm. In particular, these examples illustrate the need for disjunctions of postconditions in both the base case and inductive steps.

#### 3.3.1 Multiple postconditions and postconditions in an inductive step

The following example leads to multiple postcondition. Consider the following non-bisimilar processes: \([x = y] = [w = z] \tau \not\prec \tau \). Observe that clearly \( \tau \not\rightarrow 0 \) but \([(x = y] = [w = z] \tau \not\rightarrow 0 \) only if \( x \theta = y \theta \) or \( w \theta = z \theta \). Thus, \([x = y] = [w = z] \tau \models \langle x = y \rangle \tau \vee \langle w = z \rangle \tau \) is a distinguishing formula biased to the left process, while \( \tau \models \langle x \rangle \tau \) is biased to the right.

Now consider an example where postconditions are required in the inductive case. Firstly observe that \( \pi a + \pi b \not\prec \pi a \) are distinguished since \( \pi a + \pi b \not\rightarrow 0 \), but process \( \pi a \) can only
make a $\tilde{b}b$ transition under a substitution such that $a = b$. Hence we have the distinguishing formulae $\tilde{a}a + \tilde{b}b \models (\tilde{b}b)\mathfrak{t}$ and $\tilde{a}a \models (a = b)\mathfrak{t}$.

For the inductive case, consider $P \triangleq \tau, (\tilde{a}a + \tilde{b}b) + [x = y]\tau, \tilde{a}a \not\models \tau. (\tilde{a}a + \tilde{b}b) + \tau, \tilde{a}a \triangleq Q$. Let us lead by $Q \mathrel{\xrightarrow{\tau}} \tilde{a}a$, which can only be matched by $P \mathrel{\xrightarrow{\tau}} \tilde{a}a + \tilde{b}b$. By the above observation, we have distinguishing formulae for $\tilde{a}a + \tilde{b}b \not\models \tilde{a}a$. Furthermore, $P \theta \mathrel{\xrightarrow{\tau}}$ for substitutions $\theta$ such that $x\theta = y\theta$.

This leads to the following distinguishing formula for the left side, consisting of a box $\tau$ followed by a disjunction of the left distinguishing formula for $\tilde{a}a + \tilde{b}b \not\models \tilde{a}a$, and the postcondition for any additional $\tau$ transitions. $\tau, (\tilde{a}a + \tilde{b}b) + [x = y]\tau, \tilde{a}a \models [\tau] (\tilde{b}b)\mathfrak{t} \lor (x = y)\mathfrak{t}$.

The distinguishing formula for the right process is diamond $\tau$ followed by the right distinguishing formula for $\tilde{a}a + \tilde{b}b \not\models \tilde{a}a$, as follows: $\tau, (\tilde{a}a + \tilde{b}b) + \tau, \tilde{a}a \models [\tau][\tilde{b}b] (a = b)\mathfrak{t}$.

### 3.3.2 Formulae generated by substitutions applied to labels

In some cases substitutions applied to labels play a role when generating distinguishing formulae. For a minimal example consider the following non-bisimilar processes: $\tilde{a}a \not\models \tilde{b}b$. A distinguishing strategy is where process $\tilde{a}a$ makes a $\tilde{b}b$ transition, which cannot be matched by $\tilde{a}a$. However, $\tilde{a}a\sigma \models \tilde{b}b\sigma$ for any substitution $\sigma$, leading to distinguishing formula $\tilde{b}b\langle a = b \rangle\mathfrak{t}$ biased to $\tilde{a}a$. Notice substitution $\sigma$ is applied to both the process and the label.

For a trickier example consider the following: $\nu b. \pi b. a(x), [x = b]\pi x \not\models \nu b. \pi b. a(x). \pi x$. After two actions, the problem reduces to base case $[x = b]\pi x \not\models \nu b. \pi b. x = x. \pi x$, where $\pi x$ can perform a $\pi x$ action, but $[x = b]\pi x$ cannot. However, $([x = b]\pi x)\{b_2\} \not\models \nu b. \pi b. x = x. \pi x$ does hold, and furthermore $\{b_2\}$ respects $a^1. b^\nu . x^i$. From these observations we can construct a distinguishing formula biased to the left as follows: $\nu b. \pi b. a(x), [x = b]\pi x \models [a(b)] [a(x)] [\pi x] \langle x = b \rangle\mathfrak{t}$.

### 3.3.3 Alternative forms for distinguishing formulae

For the two non-bisimilar processes $[x = y] \tau. \tau + \tau \not\models \tau. \tau + \tau$, we can think of a distinguishing formula biased to the left process: $[x = y] \tau. \tau + \tau \models [\tau] (x = y)\mathfrak{t}$. However, this is different from the left-biased formula generated by the algorithm: $[x = y] \tau. \tau + \tau \models [\tau] (\pi \pi \lor (x = y)\mathfrak{t})$. Thus, there may exist alternative distinguishing formulae other than those generated by the algorithm.

### 3.3.4 An elaborate example demanding intuitionistic assumptions

For a more elaborate example consider the following.

\[
\tau, (\tau + \tau. \tau + \tau, [x = y][w = z]x) \triangleq P \not\models Q \triangleq \tau, (\tau + \tau. \tau + \tau, [x = y]x) + P
\]

A non-bisimilarity strategy is as follows: firstly, lead by transition $Q \mathrel{\xrightarrow{\tau}} Q'$ on the right, matched by transition $P \mathrel{\xrightarrow{\tau}} P'$; secondly, lead by $P' \mathrel{\xrightarrow{\tau}} [x = y][w = z]x$ on the left, matched in three possible ways by $Q' \mathrel{\xrightarrow{\tau}} 0$, $Q' \mathrel{\xrightarrow{\tau}} \tau$ and $Q' \mathrel{\xrightarrow{\tau}} [x = y]. \tau$. To distinguish 0 from $[x = y][w = z]x$ observe that $([x = y][w = z]x)\{z_2\} \mathrel{\xrightarrow{\tau}} 0$ but 0 can make no $\tau$ transition; hence distinguishing formulae for 0 and $[x = y][w = z]x$ are 0 $\models [\tau] \pi \pi$ and $[x = y][w = z]x \models [x = y][w = z]x$. To distinguish $[x = y]x$ observe that $([x = y]x)\{z_2\} \mathrel{\xrightarrow{\tau}} 0$, but $[x = y][w = z]x\{z_2\}$ can only make a $\tau$ transition under a substitution such that also $w = z$; hence $[x = y]x \models [x = y]x\mathfrak{t}$ and $[x = y][w = z]x$. 


z]τ |= [τ]⟨w = z⟩τ are distinguishing formulae. The same formulae also distinguish τ from [x = y]⟨w = z⟩τ. Thereby the algorithm in the completeness proof generates the following:

\[ P \models [τ]⟨x⟩(⟨[x = y]τ]⟨w = z⟩τ ∧ [x = y]⟨w = z⟩τ) \quad Q \models [τ]⟨y⟩(⟨[x = y]τ⟩τ ∨ [x = y]⟨τ⟩τ) \]

The strategy explained above is not unique. An alternative strategy can generate different distinguishing formulae: \( P \models [τ]⟨x⟩⟨[x = y]τ⟩τ ∨ [x = y]⟨τ⟩τ \) and \( Q \models [τ]⟨y⟩⟨[x = y]τ⟩τ ∧ [τ]⟨x = y⟩ττ \). Note if we assume the law of excluded middle, both processes above become equivalent to \( τ.(τ + τ.τ) \). Fortunately, we do not assume the law of excluded middle.

## 4 Related work

We consider the relationship between the intuitionistic modal logic for open bisimilarity presented in this work and established classical logics. We also compare this work to existing work claiming to characterise open bisimilarity for the \( π \)-calculus.

### 4.1 Comparison to classical logics for late bisimilarity

The late Milner-Parrow-Walker logic, called \( L\mathcal{M} \) [9] for “(L) late modality with (M) match” differs from the logic presented in this paper in three significant ways: firstly, free names are a priori assumed to be distinct; secondly, \( L\mathcal{M} \) is classical, that is, the law of excluded middle for name equalities is assumed; and thirdly the late input box modality is defined differently as follows — involving an existential quantification over substitutions:

\[ P \models L\mathcal{M}(\exists z_0); \quad Q \models L\mathcal{M}(\forall z_0) \]

To see that logical equivalence for \( L\mathcal{M} \) does not define a congruence, consider the processes \([x = y]\overline{\exists}x \) and 0. These processes satisfy the same set of late formulae (any formula equivalent to \( \overline{\exists} \)), since, for \( L\mathcal{M} \), x and y are a priori assumed to be distinct names. However, \( a(y).[x = y]\overline{\exists}x \) and \( a(y).0 \) have distinguishing formulae \( a(y).[x = y]\overline{\exists}x \models L\mathcal{M}(a(y))\overline{\exists}x \) biased to the left and its de Morgan complement \( a(y).0 \models L\mathcal{M}(a(y))\overline{\exists}x \) biased to the right.

Between open bisimilarity and late bisimilarity there is late congruence, which is the greatest congruence relation contained in late bisimilarity. Late congruence must contain open bisimilarity, since open bisimilarity is contained in late bisimilarity and open bisimilarity is a congruence. Late congruence also has a simpler characterisation: \( P \) and \( Q \) are late congruent whenever for all substitutions \( σ \), \( Pσ \) is late similar to \( Qσ \). The quantification over all substitutions, combined with the law of excluded middle, has the effect that we check late bisimilarity with respect to all combinations of equalities and inequalities between free names.

As for open bisimilarity, \([x = y]\overline{\exists}x \) and 0 are not late congruent. This is because for substitution \( \{z_0\} \) and \( \{x = y]\overline{\exists}x \) are clearly not late similar. This illustrates that late congruence is strictly finer than late bisimilarity. However, open bisimilarity is still strictly finer than late congruence, since \( τ + τ.τ + τ.τ \) are late congruent. Late congruence holds since, for any substitution \( θ \), \( (τ + τ.τ)θ \) and \( τ + τ.τ + τ.[x = y]τ \) are late similar. In contrast, we know these processes are not open bisimilar; and furthermore, have distinguishing formula that rely on the absence of the law of excluded middle.
4.2 Other embeddings into intuitionistic nominal logic

Tiu and Miller [16] studied embeddings of the π-calculus into the logic LINC, as well as late and open bisimilarity and their respectful modal logics. This is the most closely related work since our encodings in Abella were adapted from their work. In their encoding, both late and open bisimilarity are encoded by essentially the same modalities, differing only in the the law of the excluded middle for names and the quantification of free variables. However, no examples of distinguishing formulae for open bisimilarity were provided; and, critically, the proof made flawed assumptions about the existence of a syntactic negation of a formula, which we observe in this work is not permitted.

A problem with the approach of Tiu and Miller is the reuse of the input box modality from LM, which involves an existential quantification over substitutions. In contrast, our input box modality in OM involves universal quantification over all respectful substitutions. Our choice in OM is critical for generating distinguishing formulae. For example, the following processes are not open bisimilar: $a(x).\tau + a(x) + a(x).[x = a] \not\sim a(x).\tau + a(x)$.

For the above processes, the algorithm for distinguishing formulae in Proposition 14, correctly generates the following OM formula biased to the right:

$[a(x)](\langle \tau \rangle t t \lor [\tau] f f)$.  

However, using only late modalities, as in Tiu and Miller, there is no distinguishing formula for these processes biased to the right: e.g., the formula with a late modality $[a(x)]^L(\langle \tau \rangle \mathfrak{t} \lor [\tau] \mathfrak{f})$ succeeds for both processes, even when rejecting the law of excluded middle; also the formula $[a(x)]^L(\langle \tau \rangle [\tau] \mathfrak{f} \lor \langle \tau \rangle [x = a] \mathfrak{f})$ fails for both processes, despite being distinguishing in classical LM. The choice of modalities we make in OM make sense, since in open bisimilarity the choice of substitution is deferred as late as possible — possibly several transitions later.

4.3 A generic formalisation using nominal logic

Recently, Parrow et al. [12] provided a general proof of the soundness and completeness of logical equivalence for various modal logics with respect to corresponding bisimulations. The proof is parametric on properties of substitutions, which can be instantiated for a range of bisimulations. Moreover, their proof is mechanised using Nominal Isabelle. The conference version [12], sketches how to instantiate the abstract framework for open bisimilarity in the π-calculus without input prefixes only. However, we understand from communication with the authors that open bisimilarity for the π-calculus with input prefixes will be covered in a forthcoming extended version.

Stylistically, our intuitionistic modal logic is quite different from an instantiation of the abstract framework of Parrow et al. for open bisimilarity. Their framework, is classical and works by syntactically restricting “effect” modalities in formulae, depending on the type of bisimulation. Their effects represent substitutions that reach worlds permitted by the type of bisimulation. In contrast, the modalities of the intuitionistic modal logic OM in this paper are syntactically closer to long established modalities for the π-calculus [9]; differing instead in their semantic interpretation and in the absence of classical negation. An explanation for the stylistic differences is that for every intuitionistic logic, such as the intuitionistic modal logic in this work, there should be a corresponding classical modal logic based on an underlying Kripke semantics. Such a Kripke semantics would reflect the accessible worlds, as achieved by the syntactically restricted effect modalities in the abstract classical framework instantiated for open bisimilarity.
Conclusion

The main result of this paper is a sound and complete logical characterisation of open bisimilarity for the \( \pi \)-calculus. To achieve this result, we introduce modal logic \( \mathcal{OM} \), defined in Fig. 2. The soundness of \( \mathcal{OM} \) with respect to open bisimilarity, Theorem 5, is mechanically proven in Abella. The details of the completeness, Theorem 6, are provided in Section 3.

There are several novel features of \( \mathcal{OM} \) compared to established modal logics for \( \pi \)-calculus, such as \( \mathcal{LM} \) characterising late bisimilarity. Firstly, as demonstrated in Examples 2.3.1 and 3.3.4, the absence of the law of excluded middle is essential for the existence of distinguishing formulae in \( \mathcal{OM} \) for certain processes that are not open bisimilar (but are late congruent). The absence of the law of excluded middle is an intuitionistic assumption; and, as explained in the introduction, \( \mathcal{OM} \) can indeed be considered to be a conservative extension of intuitionistic logic. Furthermore, in contrast to classical modal logics such as \( \mathcal{LM} \), diamond and box modalities have independent interpretations, not dual to each other. These properties are expected under criterion set out for intuitionistic modal logics [14]. The absence of de Morgan dualities over modalities complicates the construction of distinguishing formulae.

The completeness proof involves an algorithm, Proposition 14, that constructs distinguishing formulae for non-bisimilar processes. To use this algorithm, firstly attempt to prove that two processes are open bisimilar. If they are non-bisimilar, after a finite number of steps, a distinguishing strategy, according to Def. 7, will be discovered. The strategy can then be used to inductively construct two distinguishing formulae, biased to each process. A key feature of the construction is that there are restricted versions of absolute truth by preconditions \( (\sigma \langle \pi \rangle t \text{ restricted from } \langle \pi \rangle t) \) and, dually, there are relaxed versions of absolute falsity by postconditions \( (\langle \pi \rangle \sigma t \text{ relaxed from } \langle \pi \rangle \text{ f f}) \), as demonstrated in Examples 2.2.2 and 3.3.1.

Our logic \( \mathcal{OM} \) is suitable for formal and automated reasoning; in particular, it has natural encodings in Abella for mechanised reasoning, used to establish Theorem 5, and Bedwyr [3] for automatic proof search. All bisimulations and satisfactions in examples have been automatically checked in Bedwyr and are available online: https://github.com/kyagrd/NonBisim2DF. In addition, our distinguishing formulae generation algorithm is implemented in Haskell [1].

Future work includes justifying whether or not \( \mathcal{OM} \) is complete for infinite processes with replication or recursion, as discussed around Lemma 15. A related problem is to extend \( \mathcal{OM} \) with fixed points, as in the \( \mu \)-calculus [7]. Such an extension could lead to intuitionistic model checkers invariant under open bisimulation, where the call-by-need approach to inputs is related to symbolic execution. We are also interested in extensions of \( \mathcal{OM} \) for open bisimulation in the spi-calculus [15].

Acknowledgments. We are grateful to Sam Staton for providing an example that helped us in the completeness proof.

References


Ugo Montanari and Vladimiro Sassone. Dynamic congruence vs. progressing bisimulation for CCS. Fundamenta Informaticae, 16(2), 1992.


Consistently-Detecting Monitors*

Adrian Francalanza

CS, ICT, University of Malta, Msida, Malta
adrian.francalanza@um.edu.mt

Abstract
We study a contextual definition for deterministic monitoring based on consistent detections. It is defined in terms of the observed behaviour of the monitor when instrumented over arbitrary systems. We give an alternative, coinductive definition based on controllability which does not rely on system quantifications, and show that it is fully-abstract wrt. the former definition. We then develop a symbolic counterpart to the controllability definition to facilitate an automated analysis for controllable monitors involving data.

1998 ACM Subject Classification F.3.1 Specifying and Verifying and Reasoning about Programs, F.3.2 Semantics of Programming Languages - Process models, D.2.5 Testing and Debugging

Keywords and phrases Runtime Monitoring, Deterministic Behaviour, Controllability, Compositional Reasoning, Symbolic Analysis

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2017.8

1 Introduction
Monitors are computational entities that observe the executions of other entities (referred to hereafter as systems) with the aim of accruing system information [32, 26], comparing system executions against some behavioural specification [23, 7], or reacting to the observed executions via adaptation or enforcement procedures [13, 36]. They are typically considered to be part of the Trusted Computing Base (TCB) and, consequently, their descriptions are expected to be correct. A correctness requirement often presumed of monitors is that they should exhibit deterministic behaviour. Yet, for most monitoring frameworks, such a requirement is seldom specified in unambiguous terms. In fact, there are a number of viable alternatives that one could consider (e.g., [44, 29, 25, 1]) and it is unclear how to choose one over the other in an objective manner. Moreover, these definitions often fail to account for the instrumentation mechanism used to compose a monitor with the system under scrutiny which may, in turn, affect monitoring behaviour. All of this leads to a poor understanding of what should be expected of a monitor, and may give rise to discrepancies between these expectations and what needs to be guaranteed by the monitor implementer in practice.

Non-determinism is intrinsic to a number of computational models used for expressing monitors and monitored systems. In fact, a substantial body of work on monitors is either cast in terms of inherently non-deterministic formalisms such as Büchi automata [45, 17], or formalisms that admit non-determinism such as process calculi and labelled transition systems [12, 46, 30, 22, 10, 23]. Non-deterministic computation arises naturally in concurrent and distributed programming, used increasingly for runtime monitoring [37, 24, 21, 8, 11]. Furthermore, a growing number of monitoring tools employ automata-based specification languages [16, 5, 41, 18] that offer rudimentary support for ensuring deterministic behaviour:

* Partly supported by UoM and RANNIS projects CPSRP05-04 and TheoFoMon:163406-051.
their respective implementations are either thread-unsafe \cite{41} or admit arbitrary code for transition-triggered actions \cite{14,16}.

This paper sets out to investigate deterministic behaviour for monitors. The study is limited to execution monitors (sequence recognisers) \cite{43,36}, used extensively for Runtime Verification (RV). Our work is developed in terms of an expository formalism (similar to the aforementioned work on transition-based descriptions) expressing monitored systems that can analyse trace events carrying data and admit degrees of non-determinism. We propose a contextual definition for deterministic monitor behaviour, founded on the observational behaviour that can be discerned when a monitor is instrumented to execute with any arbitrary system under scrutiny. The definition serves two purposes. First, its contextual nature allows us to admit as many correct monitors as possible, as long as these cannot be externally perceived to behave non-deterministically—we contend that the resulting definition is fairly intuitive. Second, it allows us to justify design decisions for an alternative definition describing the deterministic behaviour of monitors, based instead on the notion of controllability \cite{31}. We show a correspondence between these two definitions. In addition, we demonstrate how the alternative definition (which is arguably less intuitive than its contextual counterpart) is more amenable to automated analyses for assessing the deterministic behaviour exhibited by monitors. In particular, we study how this alternative definition can be reformulated in symbolic terms, to facilitate a tractable handling of infinite-state monitor analysis due to data.

\textbf{Example 1.} The monitor description $m_1$ accepts traces from an authenticator, that challenges (event $\text{chl}$) a supplicant for an arbitrary value $x$ followed by the supplicant’s authentication ($\text{aut}$) with the (correct) encoding of $x$, $y = \text{enc}(x)$. The authenticator subsequently acknowledges ($\text{ack}$) using the same value $y$. The guard construct $\text{chl}(x)$ quantifies over any value $x$; guards $\text{aut}(v)$ and $\text{ack}(v)$ require authentications (resp. acknowledgments) for a specific value $v$. ⊤ denotes acceptance.

$$
m_1 \triangleq \text{chl}(x), \text{let } y = \text{enc}(x) \text{ in } (\text{aut}(y).\text{ack}(y).\top)$$

$$
m_2 \triangleq \text{chl}(x), \text{let } y = \text{enc}(x) \text{ in } (\text{aut}(y).\text{ack}(y).\top + \text{aut}(z).\text{if } z \neq y \text{ then } \text{ack}(z').\bot)$$

$$
m_3 \triangleq \text{chl}(x), \text{let } y = \text{enc}(x) \text{ in } (\text{aut}(y).\text{ack}(y).\top + \text{aut}(z).\text{if } z \neq y \text{ then } \text{ack}(z').\bot \text{ else } \text{ack}(z).\top)$$

It is easy to inadvertently introduce non-determinism. Monitor $m_2$ extends $m_1$ (using the choice operator, $+$) with the intention of rejecting (note the verdict $\bot$) acknowledgments following authentications whose value is not the encoding of the challenge value $x$. When $v_2 \neq \text{enc}(v_1)$, the violating trace $t = \text{chl}(v_1).\text{aut}(v_2).\text{ack}(v_2) \ldots$ is always rejected by $m_2$. More subtly, however, when $v_2 = \text{enc}(v_1)$ the trace $t$ may cause the monitor to non-deterministically choose either branch, whereby the unintended branch does not reach a $\top$ verdict. But non-determinism is not necessarily conducive to inconsistent verdicts and, in cases where verdicts are considered as the only monitor observable behaviour, such non-determinism may be tolerated. Monitor $m_3$ deterministically accepts trace $t$ when $v_2 = \text{enc}(v_1)$, albeit along different execution paths, and deterministically rejects it whenever $v_2 \neq \text{enc}(v_1)$.

The main contributions of the paper are:

1. A contextual definition for observationally deterministic monitoring behaviour, Definition 6.
2. An alternative definition based on controllability, Definition 11, that coincides with it, Theorem 13.
Monitors

\[
\begin{array}{llll}
\text{w, o} & \in & \text{VERD} ::= & \top \quad \text{(accept)} \\
& | & & \bot \quad \text{(reject)} \\
\text{m, n} & \in & \text{MON} ::= & w \quad \text{(verdict)} \\
& | & & \text{let } x = e \text{ in } m \quad \text{(evaluate)} \\
& | & & \text{if } b \text{ then } m \text{ else } n \quad \text{(conditional)} \\
& | & & \text{rec } X.m \quad \text{(recursion)} \\
\end{array}
\]

\[
\begin{array}{llll}
\text{IrT} & [b] = \text{true} & \text{IrF} & [b] = \text{false} \\
\text{GQ} & \text{let } e = v \\
\text{AsS} & s \xrightarrow{m} n & \text{AsM} & m \xrightarrow{n} n \\
\end{array}
\]

\[\text{figure 1} \quad \text{A Model for describing Instrumented Systems.}\]

The paper is structured as follows. Section 2 presents the monitor framework used for this study, allowing us motivate our touchstone definition for deterministic monitor behaviour in Section 3. Section 4 presents a fully-abstract alternative definition that is amenable to compositional reasoning. In Sections 5 and 6 symbolic variants are developed for automation purposes. Section 7 concludes.

2 Systems, Monitors, Instrumentation and Monitored Systems

We perceive systems as entities that generate events while executing. Observable events, \( \eta \in \text{Evt} \), are those visible to a monitor and have the form \( l(v) \) where \( l \) is an event label taken from a set \( L \), \( k \in \text{Lab} \), and \( v \) is an event payload taken from some unspecified value domain \( v, u \in \text{Val} \). Events capture a number of computational notions such as inputs/outputs in message-passing programs [22], or method/function calls and returns [13, 38]. To simplify our technical development, we consider monadic (i.e., single-valued) events but the formalism can be extended to accommodate polyadicity.

Systems may be described as Labelled Transition Systems (LTSs) [2, 34], triples \( \langle \text{Sys, Act, } \rightarrow \rangle \) consisting of a set of states, \( s, r \in \text{Sys} \), a set of actions, \( \alpha, \beta \in \text{Act} = \text{Evt} \cup \{ \tau \} \) that include all observable events \( \text{Evt} \) and a distinguished silent action \( \tau \not\in \text{Evt} \) for unobservable events, and a transition relation, \( \rightarrow \subseteq (\text{Sys} \times \text{Act} \times \text{Sys}) \). The suggestive notation \( s \xrightarrow{\alpha} s' \) denotes \( (s, \alpha, s') \in \rightarrow \), whereas \( s \xrightarrow{\tau} \) denotes \( (\exists s' : s \xrightarrow{\alpha} s') \). As usual, we write \( s \Rightarrow s' \) in lieu of \( s(\tau^*) \Rightarrow s' \) and \( s \Rightarrow s' \) for \( s \Rightarrow \cdot \Rightarrow \cdots \Rightarrow s' \), referring to \( s' \) as a \( \eta \)-derivative of \( s \); \( s \xrightarrow{t} \) stands for \( \exists s' : s \xrightarrow{t} s' \). We let traces, \( t, u \in \text{Evt}^* \), range over (finite) sequences of observable events and write \( s \xrightarrow{t} \cdots \xrightarrow{s_n} s_n \) as \( s \xrightarrow{t} s_n \), where \( t = \eta_1, \ldots, \eta_n \). The notation \( u \ldots \) is occasionally used to denote the existence of some trace \( t \) with a prefix \( u \).

We presuppose an expression language \( e, d \in \text{Exp} \) that ranges over the (event) value domain \( \text{Val} \) and a denumerable set of expression variables \( x, y, z \in \text{Vars} \); \( \bar{e} \) and \( \bar{d} \) resp.
denote lists of expressions and variables. We also assume a boolean expression language $b, c \in \text{BExp}$ defined over $\text{EXP}$ that includes standard constructs for conjunctions, $b \land c$, and negations, $\neg b$, but also equality predicates over expressions $e = d$. The meta-functions $\text{fv}(e)$ and $\text{fv}(b)$ return the free variables in the resp. expressions; expressions are closed whenever $\text{fv}(e) = \emptyset$ and open otherwise, and similarly for boolean expressions. Valuations are total maps from variables to values, $\rho \in (\text{VARS} \rightarrow \text{VAL})$ whereas substitutions are partial maps from variables to expressions $\sigma \in (\text{VARS} \rightarrow \text{EXP})$; substitutions are denoted as $[\rho/\bar{x}]$, where $d[\rho/\bar{x}]$ represents the (simultaneous) substitution of all occurrences of $x_i \in \bar{x}$ in $d$ by the corresponding $c_i \in \bar{c}$. We assume an evaluation function that takes an expression and a valuation and returns a value, $[e]_{\rho} = v$. Similarly, boolean expressions have a semantic function mapping them to the boolean domain via a valuation, $[b]_{\rho} \in \{\text{true}, \text{false}\}$, where we presume the expected properties, e.g., $[(b \land c)]_{\rho} = \text{true}$ iff $[b]_{\rho} = \text{true}$ and $[c]_{\rho} = \text{true}$. We also assume a classical interpretation of boolean expressions, i.e., $[(\neg b)]_{\rho} = \text{true}$ iff $[b]_{\rho} = \text{false}$. To alleviate the presentation, we often work up to associativity and commutativity for conjunctions, treating $(\text{BExp}, \land, \text{true})$ as an abelian monoid. For closed expressions, we elide the valuation and write $[e]$ and $[b]$ for $[e]_{\rho}$ and $[b]_{\rho}$ resp. The satisfiability judgement for boolean expressions, $\text{sat}(b) \overset{\text{def}}{=} \exists \rho. [b]_{\rho} = \text{true}$, plays a central role in subsequent development.

Monitors, here defined by the syntax in Figure 1, constitute the focus of our study. They may reach two kinds of verdicts, $\text{VERD}$. Conclusive verdicts consist of acceptances, $\top$, and rejections, $.$. In addition, a monitor may also reach the inconclusive verdict, 0, a form of premature termination used when the generated system events of the monitor specification itself does not yield sufficient information so as to reach a definite conclusion. The monitor expression guard $l(e).m$ expects events with label $l$ and a payload value matching the evaluation of $e$, whereas the quantified guard $l(x).m$ allows the monitor to dynamically learn the payload of an event with label $l$. Monitors may branch (externally) depending on the events observed, $m+n$, or branch (internally) based on data predicates, if $b$ then $m$ else $n$. They may also perform internal computation themselves by evaluating expressions, let $x = e$ in $m$, or recurse, rec $X.p$, via term variables $X, Y, Z \in \text{TVARS}$. The constructs $l(x).m$ and rec $X.m$ act as binders for $x$ and $X$ resp. in $m$, inducing the usual notions of open/closed (monitor) terms. We work up to alpha-conversion of bound expression/term variables and use the shorthand if $b$ then $m$ for if $b$ then $m$ else 0 and $\tau.m$ for let $x = v$ in $m$ where $x \notin \text{fv}(m)$.

The semantics of closed monitors is also defined in terms of an LTS, via the transition rules in Figure 1: for each $m \in \text{MON}$ we have a dedicated LTS $(M, \text{ACT}, \rightarrow)$ where $M \subseteq \text{MON}$ are the monitors reachable from $m$ via transitions. The rules model the monitor analysis of observable events. Rule $\text{VER}$ describes how verdicts are irrevocable, meaning that a verdict can analyse any observable event but always transition to itself. In rule $\text{GrE}$, an expression guard $l(e).m$ only transitions to the continuation $m$ when observing an event matching the label $l$ with the payload equal to $[e]$. By contrast, a quantified guard $l(x).m$ transitions by analysing any event with label $l$, binding $x$ to the event payload $v$ in the continuation, $m^[v/x]$; see rule $\text{GrQ}$. The remaining rules are as expected where the term $m^[\text{rec}\ X.m[X]}$ denotes the term substitution of rec $X.m$ for free occurrences of $X$ in $m$.

A system $s$ instrumented with a monitor $m$ is referred to as a monitored system and denoted as $s \in m$. The semantics of monitored systems is defined by the instrumentation rules in Figure 1. We here adopt the composition relation studied in [22, 23], even though other instrumentation relations could have been used. Note that the chosen composition relation is still quite general: it is parametric wrt. the system and monitor abstract LTSs and it is largely independent of their specific language specifications, since it only requires the monitor LTS to contain an inconclusive (persistent) verdict state, 0. The instrumentation
When the monitor

This may be due to a number of reasons, such as event knowledge gaps or knowledge disagreements \[6\].

When the monitor cannot analyse the event generated \(^1\) and cannot internally transition to a state that enables it to do so (i.e., it is already stable, \(m \not\rightarrow \bar{r}_1\)), the instrumentation does not block the monitored system: instead, it allows the system to transition but aborts monitoring to the inconclusive verdict (rule Ter). System-monitor synchronisations are limited to observable events, and the specific entities can transition independently wrt. their respective internal moves (rules AsS and AsM).

**Example 2.** Monitor \(m_4\) below listens for input and output events \(in(v)\) and \(out(v)\) where the (integer) payload \(v \in \mathbb{N}\) reports the port number over which the communication operation is performed.

\[
\begin{align*}
    m_4 & \triangleq \rec X. \left( \langle \text{out}(80), \bot \rangle + \langle \text{in}(x), \text{if } x=80 \text{ then out}(81), \top \text{ else out}(x), X \rangle \right) \\
\end{align*}
\]

(1)

The monitor rejects system executions starting with an output on port 80 but accepts traces containing an input on port 80 followed by an output on port 81, preceded by an arbitrary number of input-output operations on any matching port other than 80. The execution below shows an accepted monitored computation for a system \(s\) generating the trace \(\langle \text{in}(85) \cdot \text{out}(85) \cdot \text{in}(80) \cdot \text{out}(81) \rangle\). In monitor \(m_4\), the binding on \(\text{in}(x)\) acts as a freeze-variable \([19]\) for the subsequent \(\text{out}(x)\) guard in the else branch.

\[
\begin{align*}
    s \in m_4 & \xrightarrow{\text{in}(85)} s' \in \langle \text{out}(80), \bot \rangle + \langle \text{in}(x), \text{if } x=80 \text{ then out}(81), \top \text{ else out}(x), m_4 \rangle & & \text{REC} \\
    s' \in m_4 & \xrightarrow{\text{out}(85)} s'' \in \langle \text{out}(85), m_4 \rangle & & \text{CnI+GrQ} \\
    m_4 & \xrightarrow{\text{in}(80) \cdot \text{out}(81)} s''' \in m_4 & & \text{GrQ} \\
    s''' & \xrightarrow{\text{in}(80) \cdot \text{out}(81)} s'''' \in \top & & \text{BEF}, \ldots
\end{align*}
\]

The instrumentation of Figure 1 delays system transitions to allow the monitor to internally transition to a state that can process the event. E.g., if a system \(r\) can generate event \(\text{out}(80)\), \(r \in m_4\) postpones this transition (MON and Ter cannot be applied) until \(m_4\) unfolds.

\[
\begin{align*}
    r \in m_4 & \xrightarrow{\text{out}(80)} r' \in \langle \text{out}(80), \bot \rangle + \langle \text{in}(x), \text{if } x=80 \text{ then } \ldots \rangle & & \text{AsM, MON} \\
\end{align*}
\]

Rule Ter is crucial both for allowing monitored computations to proceed when the monitor cannot analyse an event, but also to avoid unintended detections. E.g., if system \(r\) can generate the trace \(\langle \text{out}(90) \cdot \text{in}(80) \cdot \text{out}(81) \rangle\), this behaviour should still be permitted when instrumented with the monitor \(m_4\), but the behaviour should not be detected according to the description in Equation 1. After the initial unfolding of \(m_4\), Ter allows \(r\) to transition with \(\text{out}(90)\) but transitions \(m_4\) to the inconclusive state, \(\emptyset\), since neither guard \(\text{out}(80)\) nor guard \(\text{in}(x)\) can process the event.

\[
\begin{align*}
    r \in m_4 & \xrightarrow{\text{out}(90)} r' \in \langle \text{out}(80), \bot \rangle + \langle \text{in}(x), \text{if } x=80 \text{ then } \ldots \rangle & & \text{AsM, Ter}, \ldots
\end{align*}
\]

Had rule Ter been designed otherwise (leaving the monitor state unaltered when transiting with \(\text{out}(90)\)) the ensuing events \(\text{in}(80) \cdot \text{out}(81)\) would lead to the unintended acceptance of the trace.

\(^1\) This may be due to a number of reasons, such as event knowledge gaps or knowledge disagreements \([6]\).
3 Deterministic Monitoring Behaviour

In a monitored system, non-deterministic behaviour can be caused by either the system or the monitor. We focus on identifying non-determinism attributed to monitors, teasing it apart from non-determinism caused by system behaviour. This is motivated by the fact that, generally, one has limited control over the behaviour of a system under scrutiny. We target a definition that admits monitor non-determinism that is not externally observable. Concretely, we consider detections (i.e., conclusive verdicts) as the only externally visible aspect of a monitor and base our definition on the notion of deterministic detections—in applications such as RV, detections are associated with property satisfactions and violations [23, 7]. This immediately rules out a number of candidate definitions for deterministic monitor behaviour. For instance, a definition that considers a monitor $m$ to be deterministic whenever, for all systems $s$ and traces $t$, $s < m \not\Rightarrow s' < m'$ and $s < m \not\Rightarrow s' < m''$ implies $m' = m''$ is too stringent: it precludes the monitor description below (a slight modification on $m_1$ of Example 1)

$$m_5 \triangleq \text{chl}(x) \cdot ((\text{let } y = \text{enc}(x) \text{ in } \text{aut}(y).\text{ack}(y).\top) + (\text{aut}(\text{enc}(x)).\text{ack}(\text{enc}(x)).\top))$$

even though $m_5$ deterministically accepts traces of the form $\text{chl}(v_1) \cdot \text{aut}(v_2) \cdot \text{ack}(v_2)$ where $v_2 = \text{enc}(v_1)$. In fact, after an event $\text{chl}(v)$ (for some value $v$), monitor $m_5$ can reach two possible internal states, namely $(\text{let } y = \text{enc}(v) \text{ in } \text{aut}(y).\text{ack}(y).\top) + (\text{aut}(\text{enc}(v)).\text{ack}(\text{enc}(v)).\top)$ or $\text{aut}(v').\text{ack}(v').\top$ where $v' = \text{enc}(v)$. Other candidates (e.g., confluence defined over transitions [25, 40]) are either inadequate or not immediately applicable because they do not account for executions that do not lead to detections. E.g., $m_6$ (below) would not be confluent (consider event $\text{in}(81)$), even though it consistently rejects any trace with the prefix $u=\text{in}(80)$ (and consistently does not detect all the other traces).

$$m_6 \triangleq (\text{in}(80).\bot) + ((\text{in}(81).\text{out}(81).0) + (\text{in}(81).\text{out}(81).\text{in}(82).0))$$

Definition 3 (Detected Computations). The transition sequence

$$s < m \not\Rightarrow s_0 < m_0 \not\Rightarrow s_1 < m_1 \not\Rightarrow s_2 < m_2 \not\Rightarrow \ldots$$

is called a $t$-computation if it is maximal i.e., either it is infinite or it is finite and cannot be extended further using $\tau$-transitions. The $t$-computation above is called accepted whenever $\exists i \in \mathbb{N} \cdot m_i = \top$ and rejected when $\exists i \in \mathbb{N} \cdot m_i = \bot$. A detected $t$-computation is either an accepted or a rejected one.

Detected computations are indexed by their trace to allow us to partition computations according to the system behaviour exhibited at runtime, thus accounting for system non-determinism. Definition 3 also permits monitors to stabilise and reach verdicts in the trailing $\tau$-sequence following a $t$-trace.

Definition 4 (Deterministic Detection and Withholding). Monitor $m$ deterministically accepts (resp. deterministically rejects) for system $s$ along trace $t \in \text{Ev}^\tau$, denoted as $\text{da}(m, s, t)$ and $\text{dr}(m, s, t)$ resp., iff all $t$-computation from $s < m$ are accepting (resp. rejecting). Monitor $m$ deterministically detects for $s$ along $t$, $\text{dd}(m, s, t)$, whenever $\text{da}(m, s, t)$ or $\text{dr}(m, s, t)$. Monitor $m$ deterministically withholds for $s$ along trace $t$, $\text{dw}(m, s, t)$, iff no $t$-computation from $s < m$ is accepting or rejecting.
Example 5. For arbitrary system $s$, monitors $m_1$ of Example 1 and $m_5$ of Equation 2 deterministically accept traces with the prefix $t = \text{cli} (v_1) . \text{aut} (v_2) . \text{ack} (v_2)$ where $v_2 = \text{enc} (v_1)$ and deterministically withhold on all the other traces. Monitor $m_2$ deterministically rejects traces with the prefix $t$ above when $v_2 \neq \text{enc} (v_1)$ but does not deterministically detect traces with prefix $t$ when $v_2 = \text{enc} (v_1)$. For arbitrary $s$, monitor $m_3$ deterministically detects any trace with the above prefix $t$ (accepting or rejecting the trace depending on whether $v_2 = \text{enc} (v_1)$ or not) and deterministically withholds otherwise. For any system $s$, monitor $m_4$ of Example 2 satisfies $\text{dr} (m_4, s, t)$ when the trace $t$ is of the form $t = \text{out} (80) \ldots , \text{da} (m_4, s, t)$ when $t = (\text{in} (v_i) \cdot \text{out} (v_i))^+ \cdot \text{in} (80) \cdot \text{out} (81) \ldots$ for some $i \in \mathbb{N}$, and $\text{dw} (m_4, s, t)$ otherwise. Similarly, for all systems $s$, $m_6$ satisfies $\text{dr} (m, s, t)$ when $t = \text{in} (80) \ldots$ and $\text{dw} (m, s, t)$ otherwise.

For the rest of our study, monitors with deterministic behaviour are defined as consistently-detecting.

Definition 6 (Consistent Detection). Monitor $m$ consistently detects for system $s$, denoted as $\text{cd} (m, s)$ if for all traces $t$ we have $\text{dd} (m, s, t)$ or $\text{dw} (m, s, t)$. A monitor $m$ is consistently-detecting, denoted as $\text{cd} (m)$, whenever $\text{cd} (m, s)$ holds for any system $s$.

Example 7. Monitors $m_1, m_3, m_4, m_5$ and $m_6$ are consistently-detecting, but $m_2$ is not. Definition 6 does not require monitors to perform any detections. The monitor $m_7 \triangleq \text{rec} X . (\text{in} (81) . X) + (\text{in} (81) . \text{out} (81) . X)$ can consistently analyse an infinite number of traces for any $s$, $\text{cd} (m_7)$, even though it never flags.

A few comments are in order. First, Definition 6 abstracts away from the particular instances of the systems considered, the specifics of the monitor language and instrumentation mechanism used; this makes it applicable to arbitrary monitoring setups. Second, $\text{cd} (m, s)$ may be seen as requiring an ambiguity of $d=1$ from automata theory [29, 4], for the observable behaviour specified in Definition 4. Our setting is however more general, allowing for infinite states and alphabets (actions). Moreover, $\text{cd} (m)$ quantifies over all possible system compositions. Third, since Definition 6 is defined over monitored system behaviour, it allows us to assess the actual monitor behaviour at runtime. Particularly, the system quantification in $\text{cd} (m)$ accounts for any (indirect) effects of a system on the execution of a monitor.

Example 8. Whereas monitor $m_8 \triangleq \text{in} (81) . \perp$ is (trivially) consistently-detecting in the framework of Figure 1, the monitor $m_9 \triangleq \text{in} (x) . \text{if} \ x = 81 \ \text{then} \ \perp \ \text{else} \ 0$ is, perhaps surprisingly, not. Consider a (diverging) system $s$ with behaviour $s \overset{\text{in} (81)}{\longrightarrow} s' \overset{\tau}{\longrightarrow} s'$. Although $s \triangleq m_9$ can reject the $t$-computation for $t=\text{in} (81)$, another possible $t$-computation of $s \triangleq m_9$ is

\[ s \in m_9 \overset{\text{in} (81)}{\longrightarrow} (s' \triangleq \text{if} \ 81 = 81 \ \text{then} \ \perp \ \text{else} \ 0) \overset{\tau}{\longrightarrow} (s' \triangleq \text{if} \ 81 = 81 \ \text{then} \ \perp \ \text{else} \ 0) \overset{\tau}{\longrightarrow} \ldots \]

which never reaches a verdict. Therefore, we have $\neg \text{cd} (m_9)$ according to Definition 6.

Fourth, consistently-detecting monitors are not compositional, affecting the subsequent machinery.

Example 9. Although $m_8$ (from Example 8) and monitor $m_{10} = \text{in} (81) . \top$ are both consistently-detecting according to Definition 6, their composition, i.e., $m_8 + m_{10}$, is clearly not.
4 Controllability

In spite of its generality and intuitive nature, Definition 6 it is hard to automate directly as a correctness analysis. One major obstacle is the inherent universal quantification over systems and traces defining $cd(m)$. In this section, we set out to give an alternative definition for describing consistently-detecting monitors that does not suffer from these shortcomings. It is based on the notion of controllability [20, 31] which, in discrete event settings, roughly refers to the ability to steer a (passive) entity to designated terminal states via a series of admissible controls. In our case, the monitors will constitute the passive entities to be steered, whereas the monitored systems assume the controller’s role: the admissible controls are effectively the observable events in a monitoring setup that cause the monitor to transition, whereas the terminal states of interest are the conclusive verdicts. The proposed definition thus inverts the focus from how a system is monitored to how a monitor can be driven.

Before giving the actual definition, we first need to lift the technical machinery of Figure 1 to sets of monitors, $M, N \subseteq \text{Mon}$: this allows us to express the status whereby a monitor that can be in a number of potential states after being driven by a sequence of steering controls, which facilitates the analysis of non-compositional properties such as ours (see Example 9).

- **Definition 10.** A monitor-set $M$ potentially reaches a verdict $w$, $\text{pr}(M, w)$, when $\exists m \in M \cdot m \Rightarrow w$, and potentially analyses an event $\eta$, $\text{pa}(M, \eta)$, when $\exists m \in M \cdot m \nRightarrow \eta$. Function $\text{aft}(M, \eta)$ is defined as:

  $$\text{aft}(M, \eta) \overset{\text{def}}{=} \bigcup_{m \in M} \text{aft}(m, \eta)$$

  $$\text{aft}(m, \eta) \overset{\text{def}}{=} \{ n \mid m \Rightarrow \cdot \xrightarrow{\eta} n \} \cup \{ 0 \mid \exists n \cdot m \Rightarrow n \nRightarrow \text{ and } n \xrightarrow{\eta} \}$$

Intuitively, $\text{aft}(M, \eta)$ computes the set of reachable states from every $m \in M$ when it is asked by the instrumentation of Figure 1 to analyse an event $\eta$. The two conditions defining $\text{aft}(m, \eta)$ correspond to the monitored system transitions dictated by the respective rules $\text{Mon}$ and $\text{Ter}$ in Figure 1.

- **Definition 11 (Controllability).** A relation $R \subseteq \mathcal{P}(\text{Mon})$ is controllable iff for all $M \in R$:
  1. $\text{pr}(M, w)$ and $w \in \{ \top, \bot \}$ implies $M = \{ w \}$;
  2. $\text{pa}(M, \eta)$ implies $\text{aft}(M, \eta) \in R$.

  Controllability, denoted as the relation $\mathcal{C}$, is the largest controllable relation. A monitor $m$ (resp. monitor-set $M$) is said to be controllable iff $\{ m \} \in \mathcal{C}$ (resp. $M \in \mathcal{C}$).

Controllability is coinductive: to show that a monitor $m$ is controllable, i.e., $\{ m \} \in \mathcal{C}$, it suffices to provide a witness controllable relation $R$ such that $\{ m \} \in R$. Condition (i) in Definition 11 requires that if some $m \in M$ can reach a conclusive verdict, then every $m' \in M$ must be able to do so immediately, without requiring any preceding $\tau$-moves (hence $M = \{ w \}$); this rules out the possibility of inconsistent detections and, at the same time, prohibits diverging systems from interfering with the reaching of such verdicts (see Example 8). Condition (ii) in Definition 11 intuitively requires that this condition is satisfied for any event $\eta$ observed, by all the states that any $m \in M$ may transition to when analysing $\eta$. 
Example 12. We can show that \( m_6 \) (Equation 3) is controllable via the controllable relation \( \mathcal{R}_1 \) below:

\[
\mathcal{R}_1 = \left\{ \{m_6\}, \{\top\}, \{0, \text{in}(82), 0\}, \{0\}, \{\text{out}(81), 0, \text{out}(81), \text{in}(82), 0\} \right\}
\]

\[
\mathcal{R}_2 = \left\{ \{m_7\}, \{m_7, \text{out}(81), m_7\}, \{0, m_7\}, \{0, m_7, \text{out}(81), m_7\}, \{0\} \right\}
\]

Note that \( \{m_6\} \in \mathcal{R}_1 \). We can also finitely determine that the recursive monitor \( m_7 \) (Example 7) is controllable via the relation \( \mathcal{R}_2 \). The reader may want to check that \( \mathcal{R}_2 \) is controllable. For instance, \( \text{aft}((\{m_7, \text{out}(81), m_7\}, \text{in}(81))) = \{0, m_7, \text{out}(81), m_7\}, \text{aft}((\{m_7, \text{out}(81), m_7\}, \text{out}(81))) = \{0, m_7\} \) and, importantly, \( \text{aft}((\{0, m_7, \text{out}(81), m_7\}, \text{in}(81))) = \{0, m_7, \text{out}(81), m_7\} \) itself.

Controllability coincides with Definition 6: we can use Definition 11 as a sound and complete proof technique to determine whether a monitor \( m \) satisfies \( \text{cd}(m) \), side-stepping universal quantifications over systems.

Theorem 13 (Consistent Detection Full Abstraction). \( \text{cd}(m) \iff \{m\} \in \mathcal{C} \)

Example 14. As a result of Theorem 13, we can show that \( m_6 \) and \( m_7 \) are consistently-detecting via the controllable relations \( \mathcal{R}_1 \) and \( \mathcal{R}_2 \) of Example 12. We can also indirectly show that \( \neg \text{cd}(m_8 + m_{10}) \) from Example 9, by arguing that there cannot be a controllable relation \( \mathcal{R} \) with \( \{m_8 + m_{10}\} \in \mathcal{R} \). For suppose that such an \( \mathcal{R} \) exists. By Definition 11(ii) the monitor-set \( \text{aft}((\{m_8 + m_{10}\}, \text{in}(81))) = \{\top, \bot\} \) must also be in \( \mathcal{R} \); this, in turn, would necessarily mean that \( \mathcal{R} \) is not controllable since \( \{\top, \bot\} \) violates Definition 11(i).

5 Symbolic Controllability

Controllability, Definition 11, is still not adequate for a fully automated analysis of consistently-detecting monitors. Particularly, whenever the resp. event value domain is infinite, quantified guards induce an infinite number of transitions, e.g., \( l(x).m \) generates a transition with the label \( l(v) \) for every \( v \in \text{Val} \) (see rule \( \text{GrQ} \)). As a result, condition Definition 11(ii) may require the monitor analysis to consider a potentially infinite number of monitor-set states whenever monitor descriptions use quantified guards.

To this end, we define a symbolic semantics over both open\(^2\) and closed monitor terms using symbolic events, \( \theta \in \text{SEvt} \). These are similar to the events of Section 2 except that they carry variables instead of values as payloads, \( l(x) \). Symbolic transitions, \( m \xrightarrow{\mu} n \) are defined by the rules in Figure 2, where \( \mu \in \text{SEvt} \cup \{\tau\} \) ranges over both symbolic and \( \tau \) events, and the boolean expression \( b \) records the condition under which the LTS action may take place. For instance, the term \( \text{rec} X.m \) may unfold in all circumstances (i.e., \( b = \text{true} \) in rule \( \text{SRec} \)) whereas the term if \( b \) then \( m \) else \( n \) can either \( \tau \)-transition to \( m \) when \( b \) holds, or to \( n \) when the converse, \( \neg b \), holds (rules \( \text{SIfT} \) and \( \text{SIfF} \)). The other key rules in Figure 2 are \( \text{SGTE} \) and \( \text{SGQ} \): the former transitions with a symbolic event \( l(x) \) under the condition \( x=e \), whereas the latter transitions with a similar symbolic event under any circumstance. Figure 2 also defines rules for weak symbolic transitions, \( m \xrightarrow{\theta} n \), and reductions, \( m \Rightarrow n \),

\(^2\) Open wrt. expression variables \( x, y, \ldots \in \text{Vars} \) not term variables \( X, Y, \ldots \in \text{TVars} \).
where both relations aggregate boolean constraints via conjunctions. Note that weak symbolic transitions describe transition sequences where $\tau$-transitions must precede the (final) symbolic event transition. The predicate $m \xrightarrow{\not\tau} b$ denotes $\not\exists b, n \cdot m \xrightarrow{\not\tau} n$ whereas $m \xrightarrow{\tau} b$ stands for $\exists n \cdot m \xrightarrow{\rho} n$.

A constrained monitor-set $⟨b, M⟩$ is a tuple where every $m \in M$ may be open, and $b$ is a condition constraining free variables in $M$. Every $⟨b, M⟩$ abstractly represents a (potentially infinite) set of closed monitor-sets for every valuation $\rho$ satisfying $b$,

$$\{ m\rho \mid \rul{b\rho} = \text{true} \text{ and } m \in M \} \quad \text{(4)}$$

In this sense, the monitor-sets in Section 4 are special cases of constrained monitor-sets where $b = \text{true}$ and $M$ is closed. Note that whenever $\lnot\text{sat}(b)$, the constrained monitor-set $⟨b, M⟩$ denotes the empty set of monitor-sets, $\emptyset$, which is trivially controllable by Definition 11. We lift functions such as that for free variables $\text{fv}(–)$ to constrained monitor-sets in the obvious manner, e.g., $\text{fv}(⟨b, M⟩) \overset{\text{df}}{=} \text{fv}(b) \cup \text{fv}(M)$.

Example 15. The constrained monitor-set $⟨x \geq 3, \{ \text{if } x = 2 \text{ then } \top \text{ else } \bot \} \rangle$ abstractly describes all monitor-sets $\{ \text{if } x = 2 \text{ then } \top \text{ else } \bot \} \rho$ where $\rho(x) \geq 3$. For any such $\rho$, no monitor of the form $\{ \text{if } x = 2 \text{ then } \top \text{ else } \bot \} \rho$ can transition to a $\top$ verdict according to the concrete semantics of Figure 1. Symbolically, this may be expressed as $\lnot\text{sat}(x \geq 3 \lor x = 2)$.

We abstractly model controllability, Definition 11, in terms of constrained monitor-sets, the symbolic semantics of Figure 2 and the satisfiability judgement $\text{sat}(b)$ defined earlier in Section 2.

Definition 16. A constrained monitor-set $⟨b, M⟩$ potentially reaches a verdict $w$, denoted as $\text{spr}(⟨b, M⟩, w)$, whenever $\exists m \in M, c \cdot m \xrightarrow{\rho} w$ and $\text{sat}(b \land c)$. Moreover, $⟨b, M⟩$ potentially analyses a symbolic event $\theta$ along $c$, denoted as $\text{spa}(⟨b, M⟩, \theta, c)$, whenever $\exists m \in M \cdot m \xrightarrow{\theta} n$ and $\text{sat}(b \land c)$.

Defining the symbolic counterpart to $\text{aft}(M, \eta)$ of Definition 10 is less straightforward. Intuitively, from all the valuations $\rho$ satisfying (and represented by) $b$ in $⟨b, M⟩$, only a subset of them may satisfy the condition $c$ in a potentially-analyses judgement $\text{spa}(⟨b, M⟩, \theta, c)$ from Definition 16. A correct modelling of Definition 11 therefore requires us to take this fact into account.

Example 17. Consider the constrained monitor-set $⟨\text{true}, M⟩$ where $M = \{ m_{10}, m_{11} \}$ and

$$m_{10} \overset{\Delta}{=} \text{if } x = 2 \text{ then } k(1).\top \text{ else } k(1).\bot \quad m_{11} \overset{\Delta}{=} \text{if } x \leq 1 \lor x \geq 3 \text{ then } k(1).\bot \text{ else } k(1).\top$$

It turns out that for any $\rho$ satisfying $\text{true}$, $M \rho$ is controllable. Therefore, for the judgement $\text{spa}(⟨\text{true}, M⟩, k(y), (x=2 \land y=1)\rangle)$ of Definition 16, which holds since $m_{10} \overset{k(y)}{\xrightarrow{x = 2 \land y = 1}}$ and $\text{sat}(\text{true} \land x = 2 \land y = 1)$, the reachable states to be considered by a corresponding symbolic analysis (modelling Definition 11(iii)) should not include the residual state $\bot$, even though it may be reached after the event $k(y)$ with $y = 1$ (see rule $\text{SGiE}$). The reason for this is that the conditions required to symbolically reach this state, i.e., $\lnot((x = 2) \land y = 1)$ or $((x \leq 1 \lor x \geq 3) \land y = 1)$, cannot be satisfied by any $\rho$ that also satisfies the $\text{spa}(–)$ condition $⟨x = 2 \land y = 1⟩$. Symbolically, this may be expressed as $\lnot\text{sat}((x = 2 \land y = 1) \land ((x = 2 \land y = 1)))$ and $\lnot\text{sat}((x = 2 \land y = 1) \land ((x = 1 \lor x = 3) \land y = 1))$.
Symbolic Transitions

\[
\begin{align*}
\text{sVer} & \quad w \xrightarrow{\tau_{\text{true}}} w \\
\text{sTr} & \quad \text{if } b \text{ then } m \text{ else } n \xrightarrow{\tau_b} m \\
\text{sFr} & \quad \text{if } b \text{ then } m \text{ else } n \xrightarrow{\tau_{\neg b}} n
\end{align*}
\]

\[
\begin{align*}
\text{sRec} & \quad \text{rec } X. m \xrightarrow{\tau_{\text{true}}} m_{[\text{rec } X. m/X]} \\
\text{sCtrl1} & \quad m \xrightarrow{\mu_b} m' \\
\text{sCtrl2} & \quad n \xrightarrow{\mu_b} n' \\
\text{sGre} & \quad l(e). m \xrightarrow{\mu_{e=z}} m
\end{align*}
\]

Weak Symbolic Transitions and Reductions

\[
\begin{align*}
\text{sWTr1} & \quad m \xrightarrow{\theta_b} m' \\
\text{sWTr2} & \quad m \xrightarrow{\tau_b} m' \\
\text{sWr1} & \quad m \xrightarrow{\theta_b \lor c} m'' \\
\text{sWr2} & \quad m \xrightarrow{\tau_b} m' \\
\text{sWr3} & \quad m \xrightarrow{\theta_b \land c} m''
\end{align*}
\]

\[\text{let } x = e \text{ in } m \xrightarrow{\tau_{e=x}} m[x/e]\]

**Figure 2** A Symbolic Semantics for Monitors.

The complications elicited in Example 17 are even more intricate. For instance, for a particular judgement \(\text{spa}(b, M, \theta, c)\), one could have some \(m_1, m_2 \in M\) whereby \(m_1 \xrightarrow{\theta_i} m'_i\) and \(\text{sat}(b \land c \land c_i)\) for \(i \in \{1, 2\}\), but at the same time having \(c_1\) and \(c_2\) being incompatible with one another, i.e., \(\neg \text{sat}(c_1 \land c_2)\). In such cases, the respective residual states \(m'_1\) and \(m'_2\) should be analysed separately.

**Definition 18.** The relevant conditions for a monitor-set \(M\) wrt. a symbolic event \(\theta\) are:

\[
\text{rc}(M, \theta) \overset{\text{def}}{=} \{ c \mid \exists m \in M \cdot (m \xrightarrow{\theta_c} m' \text{ or } \exists n \cdot (m \Rightarrow n \text{ and } n \xrightarrow{\neg \theta_c} n')) \}
\]

The satisfiability combinations for a condition-set \(\{c_1, \ldots, c_n\}\) wrt. a condition \(b\) are:

\[
\text{sc}(b, \{c_1, \ldots, c_n\}) \overset{\text{def}}{=} \{ \{b, c'_1, \ldots, c'_n\} \mid \forall i \in 1..n \cdot (c'_i = c_i \text{ or } c'_i = \neg c_i) \}
\]

The reachable constrained monitor-sets from \((b, M)\) after \(\theta\) with condition \(c\) are:

\[
\begin{align*}
\text{saft}(b, M, \theta, c) & \overset{\text{def}}{=} \{ \langle \land B, \text{saft}(M, B, \theta) \rangle \mid B \in \text{sc}(b, c, \text{rc}(M, \theta)) \text{ and } \text{sat}(\land B) \} \\
\text{saft}(M, B, \theta) & \overset{\text{def}}{=} \{ n \mid \exists m \in M, c \cdot \text{sat}((\land B) \land c) \text{ and } (m \xrightarrow{\theta_c} n \text{ or } \exists n' \cdot m \Rightarrow n' \xrightarrow{\neg \theta_c} n' \text{ and } n' = 0) \}
\end{align*}
\]

In Definition 18, the relevant conditions for \(M\) wrt. \(\theta\), denoted as \(\text{rc}(M, \theta)\), are all the symbolic conditions that need to be considered to assess the reachable states from \(M\) for the symbolic event \(\theta\) — they are the symbolic counterpart to the transition sequences defining \(\text{af}(m, \eta)\) in Definition 10. The satisfiability combinations of a condition-set \(B\) wrt. a condition \(b\), denoted as \(\text{sc}(b, B)\), capture the maximal condition subsets in \(B\) that any valuation \(\rho\) satisfying condition \(b\) also satisfies. Every condition set \(B'\) returned by \(\text{sc}(b, B)\)
Consistently-Detecting Monitors

contains \( b \) itself and one condition \( c' \) for every boolean condition \( c \in B \) (either \( c \) itself or its negation); these combination sets partition all the valuations \( \rho \) satisfying \( b \). Symbolic reachability for \( \langle b, M \rangle \) after \( \theta \) with condition \( c \), \( \text{saft}(\langle b, M \rangle, \theta, c) \) in Definition 18, is defined wrt. all the satisfiability combinations \( B \) of \( \text{rc}(M, \theta) \) for the (fixed) condition \( b \wedge c \). Although \( \text{se}(b \wedge c, \text{rc}(M, \theta)) \) partitions all the \( \rho \) satisfying \( b \wedge c \), some of these partitions are empty. Accordingly, \( \text{saft}(\langle b, M \rangle, \theta, c) \) only considers the non-empty partitions via the satisfiability condition \( \text{sat}(\wedge B) \), where \( \wedge B \) returns the syntactic conjunction formula \( c_1 \wedge \ldots \wedge c_n \) for a boolean set \( B = \{c_1, \ldots, c_n\} \).

It is worth remarking that the symbolic LTS of Figure 2, is image-finite [42], and thus finitely branching when considering the \( \tau \)-transition graph of a term \( m \). By König’s Infinity Lemma [33] the set of constraints \( \{c \mid m \not\equiv_c \) or \( \exists n \cdot (m \Rightarrow n \text{ and } n \not\Rightarrow \) and \( n \not\Rightarrow \} \) must be finite and, as a result, \( \text{rc}(M, \theta) \) is finite too for a finite monitor-set \( M \). This ensures that \( \text{saft}(\langle b, M \rangle, \theta, c) \) is well-defined.

**Definition 19 (Symbolic Controllability).** The relation \( \mathcal{S} = (\mathsf{BExp} \times \mathcal{P}(\text{MON})) \) is called a symbolically-controllable relation iff for all constrained monitor-sets \( \langle b, M \rangle \in \mathcal{S} \):
1. \( \text{spr}(\langle b, M \rangle, w) \) and \( w \in \{\top, \bot\} \) implies \( M = \{w\} \);
2. \( \text{spa}(\langle b, M \rangle, l(x), c) \) where \( \text{frsh}(\text{fv}(\langle b, M \rangle)) = x \) implies \( \text{saft}(\langle b, M \rangle, l(x), c) \in \mathcal{S} \).

**Symbolic Controllability**, denoted as \( \mathcal{C}_{\text{sym}} \), is the largest symbolically-controllable relation. A (closed) monitor \( m \) is symbolically-controllable iff \( \langle \text{true}, \{m\} \rangle \in \mathcal{C}_{\text{sym}} \).

The clause Definition 19(ii) assumes a function \( \text{frsh}(V) \) that (deterministically) returns the next fresh variable \( x \) that is not in the variable set \( V \). When compared to Definition 11(ii), this allows us to just consider one (symbolic) event, \( l(x) \), for a finite set of constraints, as opposed to a potentially infinite set of events, i.e., \( l(v) \) for every \( v \in \text{Val} \).

**Example 20.** Recall \( m_{10} \) and \( m_{11} \) from Example 17. The monitor \( m_{12} \triangleq l(x).m_{10} + l(x).m_{11} \) can be shown to be symbolically-controllable via the relation \( \mathcal{S}_1 \) defined below, where \( b_1 = (x=2y=1), b_2 = (\neg(x=2)\wedge y=1), b_3 = ((x \leq 3)\wedge y=1) \) and \( b_4 = (\neg(x \leq 3)\wedge y=1) \); these are obtained from the relevant conditions \( \text{rc}((m_{10}, m_{11}), k(y)) = \{b_1, b_2, b_3, b_4\} \).

\[
\mathcal{S}_1 = \left\{ \langle \text{true}, \{m_{12}\} \rangle, \langle \text{true}, \{m_{10}, m_{11}\} \rangle, \langle (\text{true} \wedge b_1) \wedge b_2 \wedge b_3 \wedge b_4, \{\top\} \rangle, \langle (\text{true} \wedge b_1) \wedge b_2 \wedge b_3 \wedge b_4, \{\bot\} \rangle, \langle (\text{true} \wedge b_3) \wedge b_2 \wedge b_3 \wedge b_4, \{\bot\} \rangle \right\}
\]

For illustrative purposes, we do not simplify the constraints in the constrained monitor-sets of \( \mathcal{S} \) to show how these are derived. E.g., \( \langle (\text{true} \wedge b_1) \wedge b_2 \wedge b_3 \wedge b_4, \{\top\} \rangle \) is obtained as a result of \( \text{saft}((\text{true}, \{m_{10}, m_{11}\}), l(x), \text{true} \wedge b_3) \). In fact, the combination \( \langle (\text{true} \wedge b_1), b_1, \neg b_2, \neg b_3, b_4 \rangle \) is the only satisfiable condition-set and all the others are filtered out by \( \text{saft}((\text{true}, \{m_{10}, m_{11}\}), l(x), \text{true} \wedge b_3) \).

Symbolic Controllability, Definition 19, is sound and complete wrt. Controllability, Definition 11.

**Theorem 21 (Controllability Full Abstraction).** \( \{m\} \in \mathcal{C} \) iff \( \langle \text{true}, \{m\} \rangle \in \mathcal{C}_{\text{sym}} \).

**Example 22.** Recall \( m_3 \triangleq \text{ch}(x).m_3' \) from Example 1, recast in terms of \( m_3' \) defined below:

\[
m_3' \triangleq \text{let } y = \text{enc}(x) \text{ in } \langle \text{aut}(y).\text{ack}(y), \top + \text{aut}(z).\text{if } z \neq y \text{ then } \text{ack}(z).\bot \text{ else } \text{ack}(z).\top \rangle
\]

Example 7 stated that \( m_3 \) is consistently-detecting. This fact is hard to determine using Definition 6, whereas analyses using Definition 11 are complicated by quantifications over
the values of events. By Theorems 13 and 21, we can show that $m_3$ is consistently-detecting via the symbolic controllability relation:

$$\mathcal{S}_2 = \left\{ \langle \text{true}, \{m_3\} \rangle, \langle \text{true}, m_4' \rangle, \langle z = \text{enc}(x), \{\text{ack}z(\text{enc})(x).\top, \text{if } z \neq \text{enc}(x) \text{ then } \text{ack}(z').\bot, \text{else } \text{ack}(z).\top \} \rangle, \langle \{z = \text{enc}(x)\} \land (w = z) \land (w = \text{enc}(x)), \{\top}\rangle, \langle \langle z = \text{enc}(x) \rangle, \langle \text{if } z \neq \text{enc}(x) \text{ then } \text{ack}(z').\bot, \text{else } \text{ack}(z).\top \rangle \rangle, \langle \langle z = \text{enc}(x) \rangle, \{\bot\} \rangle \right\}$$

In $\mathcal{S}_2$ and the ensuing discussion, we alleviate our presentation by simplifying the boolean conditions used, e.g., we simply write $(z = \text{enc}(x))$ in lieu of $(\text{true} \land (z = \text{enc}(x)))$. We highlight a few points.

First, consider the second constrained monitor-set in $\mathcal{S}_2$, namely $\langle \text{true}, m_4' \rangle$. Since the semantics of Figure 2 allows expression guards and quantified guards to transition with the same symbolic event (albeit with different conditions) we are able to consider the resp. continuations in unison for the event $\text{aut}(z)$. Concretely, according to Definition 19(ii), for $\text{spa}((\text{true}, m_4'), \text{aut}(z), (z=\text{enc}(x)))$ generated by the expression guard weak transition of $m_4'$, we need to ensure that the resulting monitor-set $\langle z = \text{enc}(x), \text{saft}(\{m_4'\}, \{z = \text{enc}(x), \text{true}\}, \text{aut}(z)) \rangle$ (which evaluates to the third constrained monitor-set in $\mathcal{S}_2$) is also in the symbolic relation. At the same time, for $\text{spa}((\text{true}, m_4'), \text{aut}(z), \text{true})$ generated by the quantified guard weak transition of $m_3'$, we need to ensure that two monitor-sets are in $\mathcal{S}_2$, namely $\langle z = \text{enc}(x), \text{saft}(\{m_4'\}, \{z = \text{enc}(x), \text{true}\}, \text{aut}(z)) \rangle$ (as before) but also the constrained monitor-set $\langle \langle z = \text{enc}(x) \rangle, \text{saft}(\{m_4'\}, \langle \neg (z = \text{enc}(x)), \text{true}\rangle, \text{aut}(z)) \rangle$ (which evaluates to the fifth constrained monitor-set in $\mathcal{S}_2$).

The second point we highlight about $\mathcal{S}_2$ concerns its third constrained monitor-set. In particular, the left branch of the conditional term in this set, namely $\text{ack}(z').\bot$ in the term if $z \neq \text{enc}(x)$ then $\text{ack}(z').\bot$, is not considered by our analysis since its condition, $z \neq \text{enc}(x)$, is incompatible with the constraining condition of the monitor-set, i.e., $\neg \text{sat}(\{z = \text{enc}(x)\} \land (z \neq \text{enc}(x)))$.

Third, we also note how the condition aggregation mechanism for the consecutive symbolic events $\text{aut}(z)$ and $\text{ack}(w)$ — transferring us from the second constrained monitor-set, $\langle \text{true}, m_4' \rangle$, to the fourth, $\langle z = \text{enc}(x) \rangle \land (w = z) \land (w = \text{enc}(x)), \{\top\} \rangle$, via the third constrained monitor-set in $\mathcal{S}_2$ — enables us to symbolically relate the expression guards in $m_3$, which impose a condition such as $(z = \text{enc}(x))$ upon transition, with the quantified guard that imposes the same condition after the transition (by means of a conditional branch in its continuation). We leave it up to the interested reader to check that the remaining monitor-sets in $\mathcal{S}_2$ satisfy the conditions required by Definition 19.

---

6 On Automating Symbolic Controllability

Despite its merits, a direct implementation of the symbolic controllability from Definition 19 still would not perform well for certain recursive monitor descriptions, as shown in the following example.

- **Example 23.** Recall monitor $m_4$ from Example 2. To show that it is controllable, we need to exhibit a symbolic relation that includes $\langle \text{true}, \{m_4\} \rangle$. For some fresh variable $x$ where $\text{frsh}(\text{fv}(\langle \text{true}, \{m_4\} \rangle)) = x$, since the judgement $\text{spa}((\text{true}, \{m_4\}), \text{in}(x), \text{true})$ holds, this relation needs to include the ensuing monitor-set $\langle \text{true}, \{m_4\} \rangle$ as well, where $m_4' \triangleq \text{if } x = 80 \text{ then } \text{out}(81).\top \text{ else } \text{out}(x).m_4$. In turn, since $\text{spa}((\text{true}, \{m_4\}), \text{out}(y), (\neg (x = 80) \land y = x))$ (where $\text{frsh}(\text{fv}(\langle b, \{m_4\} \rangle)) = y$), the symbolic relation must also contain $(\neg (x = 80) \land y = x, \{m_4\})$. We thus reach the original monitor set $\{m_4\}$ but with a stronger condition, namely
\[\neg(x = 80) \land y = x.\] By extension of this reasoning, it is not hard to see that the symbolic relation required by Definition 19 needs to be infinitely large.

The problem exhibited by Example 23 is that the condition aggregating mechanism of Definition 19 does not specify any means for consolidating the boolean condition \(b\) constraining a monitor set \(M\) in \(\langle b, M \rangle\), i.e., a form of garbage collection of redundant conditions. For instance, in the constrained monitor-set \((\neg(x = 80) \land y = x, \{m_4\})\) of Example 23, the condition \((\neg(x = 80) \land y = x)\) plays no effective role in constraining the free variables in \(\{m_4\}\), of which there are none. We therefore optimise Definition 19 in a sound (and complete) manner by taking into consideration boolean sub-conditions that can be isolated and discarded. This leads to an improved automated analysis for consistently-detecting monitors.

**Definition 24 (Optimised Symbolic Controllability).** The consolidation of a boolean expression \(b\) wrt. a variable set \(V\), denoted as \(\text{cns}(b, V)\), is defined as:

\[
\text{cns}(b, V) \overset{\text{def}}{=} b_1 \text{ whenever } \text{prt}(b, V) = \langle b_1, b_2 \rangle \text{ for some } b_2
\]

where the boolean expression partitioning operation \(\text{prt}(b, V)\) is defined as:

\[
\text{prt}(b, V) \overset{\text{def}}{=} \begin{cases} 
(b_1, b_2) & \text{if } \text{sat}(b) \text{ and } b = b_1 \land b_2 \text{ and } (\text{fv}(b_1) \subseteq V) \text{ and } (V \cap \text{fv}(b_2) = \emptyset) \\
(b, \text{true}) & \text{otherwise}
\end{cases}
\]

Let the optimised symbolic reachability from \(\langle b, M \rangle\) for \(\theta\) and \(c\), \(\text{osaft}(\langle b, M, \theta, c \rangle)\), be defined as:

\[
\text{osaft}(\langle b, M, \theta, c \rangle) \overset{\text{def}}{=} \\{ (\text{cns}(\land B, V), \text{saft}(M, \land B, \theta)) \mid B \in \text{sc}(b \land c, \text{rc}(M, \theta)) \text{ and } \text{sat}(\land B) \}
\]

A relation \(S \subseteq (\text{BExp} \times \mathcal{P}(\text{MON}))\) is called optimised symbolically-controllable iff for all \(\langle b, M \rangle \in S\):

1. \(\text{spr}(\langle b, M, w \rangle)\) and \(w \in \{\top, \bot\}\) implies \(M = \{w\}\);
2. \(\text{spa}(\langle b, M, l(x), c \rangle)\) where \(\text{frsh}(\text{fv}(\langle b, M \rangle)) = x\) implies \(\text{osaft}(\langle b, M, l(x), c \rangle) \subseteq S\).

The largest optimised symbolically-controllable relation is denoted by \(C_{\text{sym}}^{\text{opt}}\). A (closed) monitor \(m\) is said to be optimised symbolically-controllable iff \(\langle \text{true}, \{m\} \rangle \in C_{\text{sym}}^{\text{opt}}\).

We highlight the salient points from Definition 24. First, boolean consolidation in a constrained monitor-set, \((\text{cns}(b), M)\), should not change the set of concrete monitor sets represented by \(\langle b, M \rangle\) and, for this reason, we cannot consolidate unsatisfiable boolean conditions. For instance, even if \(x \not\in \text{fv}(M)\), it is still unsound to optimise \(\langle \text{true} \land x \not= x, M \rangle\) to \(\langle \text{true}, M \rangle\) based on the fact that \(\neg \text{sat}(x \not= x)\). Concretely, from Equation 4 of Section 5 we know that \(\langle \text{true} \land x \not= x, M \rangle\) denotes the empty set of monitor-sets, \(\emptyset\), whereas \(\langle \text{true}, M \rangle\) represents the set \(\{m \in \mathcal{M} : [m] = \text{true} \text{ and } m \in M\}\). Second, consolidation should ideally filter out as much redundant constraints as possible, e.g., in \(\langle b_1 \land b_2, M \rangle\) we should remove \(b_2\) whenever \(\text{fv}(b_2) \cap \text{fv}(M) = \emptyset\). In Definition 24 we require the strongest possible condition for the residual condition \(b_1\) in \(\langle b_1 \land b_2, M \rangle\), i.e., \(\text{fv}(b_1) \subseteq \text{fv}(M)\), which indirectly implies that the resp. condition variables are partitioned \(\text{fv}(b_1) \cap \text{fv}(M) = \emptyset\). This partitioning is crucial for a sound consolidation, e.g., in \(\langle \neg((x = 80) \land y = x), M \rangle\), it is unsound to just remove the subcondition \(\neg(x = 80)\) when \(x \not\in \text{fv}(M)\) and \(y \in \text{fv}(M)\). Although \(\text{prt}(b, V)\) can be refined further (while still observing core requirements for soundness such as variable condition partitioning), in Definition 24 we opted for a less elaborate condition that suffices our exposition. Third, we highlight the fact that the conditions specifying \(\text{cns}(b)\) in Definition 24 yield a unique consolidated condition up to semantic equivalence meaning that, in an implementation of the framework, this can be defined as a function.
Theorem 25 (Optimised Controllability). \((\text{true}, \{m\}) \in C^{\text{opt}} \iff (\text{true}, \{m\}) \in C^{\text{sym}}\)

Example 26. As a result of Theorems 13, 21 and 25, we can show that \(m_4\) of Example 2 is consistently-detecting by exhibiting the optimised symbolic relation \(S_3\) below. For concreteness, we show how the constrained boolean expressions are calculated. In particular, the first constrained monitor-set in \(S_3\) denotes both the starting pair \((\text{true}, \{m_4\})\), but also the pair \((\text{true}, \{m_4\})\).

\[S_3 = \left\{ \left\langle \text{true}, \{m_4\} \right\rangle, \left\langle \text{true}, \{m_4\} \right\rangle, \left\langle \text{true}, \{m_4\} \right\rangle \right\}\]

The interested reader may check that the conditions of Definition 24 are satisfied by \(S_3\).

Using standard techniques [42, 3], an algorithm constructing symbolically-controllable relations from Definition 24 can be extracted more easily. Moreover, the completeness aspect in Theorems 13, 21, and 25 should enable such an automation to infer a counter-example (system and trace) from failed attempts, thereby explaining why a monitor is not consistently-detecting.

Example 27. Recall monitor \(m_2\) from Example 1. Assuming the shorthand abbreviations \(m_2 = \text{aut}(y) \cdot \text{ack}(y) \cdot \top + \text{aut}(z) \cdot (\text{if } z \neq y \text{ then } \text{ack}(z') \cdot \bot)\), and 

\[M = \text{if } z \neq \text{enc}(x) \text{ then } \text{ack}(z') \cdot \bot \cdot \text{ack} (\text{enc}(x)) \cdot \top,\]

compiling a relation satisfying Definition 24 fails because it needs to include:

\[\langle \text{true}, \{\text{let } y = \text{enc}(x) \text{ in } m_2\} \rangle \quad \text{since} \quad \text{spa}(\text{true}, \{m_2\}, \text{chl}(x), \text{true})\]

\[\langle z = \text{enc}(x), M \rangle \quad \text{since} \quad \text{spa}(\text{true}, \{\text{let } y = \text{enc}(x) \text{ in } m_2\}, \text{aut}(z), z = \text{enc}(x))\]

\[\langle \text{true}, \{\top, 0\} \rangle \quad \text{since} \quad \text{spa}(z = \text{enc}(x), M), \text{ack}(w), (w = \text{enc}(x))\]

The final pair \((\text{true}, \{\top, 0\})\) violates Definition 24(i) and, via the symbolic events on right-hand column containing the \(\text{spa}(\_\_\_)\) assertions that lead to it, one can construct the counter-example inducing the inconsistent detection, \(i.e.,\) a system \(s\) producing the trace \(\text{chl}(v) \cdot \text{aut}(u) \cdot \text{ack}(u)\) where \(u = \text{enc}(v)\).

7 Conclusion

Monitors should provide guarantees that they will operate correctly when instrumented with any system [35]. At the same time, for this requirement to be scalable, the corresponding correctness analysis that determines it must also be compositional: monitors should be verified separately, independent of the systems they may be instrumented with. The fact that monitors tend to be considerably smaller (in size and complexity) than the systems they observe further justifies this point. This paper provides two definitions that formalise deterministic monitoring behaviour, Definition 6 (consistently-detecting monitors) and Definition 11 (controllable monitors), that address these seemingly conflicting concerns; it also shows that the two definitions coincide, Theorem 13. In addition, the paper also studies alternative definitions for controllability, Definitions 19 and 24, that enable the implementation of sound and complete symbolic analyses, Theorems 21 and 25.

Our methods provide a systematic way for factoring out auxiliary reasoning on data from the analysis relating to the branching structure of the monitors; the former kind of reasoning can be determined by calling on an independent satisfiability solver. In fact, for the specific case of our expository monitor language in Figure 1, one can show that our methods yield finite symbolic transition graphs, making the latter reasoning decidable modulo the expression and boolean language used. The results obtained in this work should also be general enough
Consistently-Detecting Monitors

to be applicable to other monitoring systems. For instance, Definitions 6, 11, 19, and 24 are independent of the kind of systems monitored, the syntax of the monitor language, and the event value domains and expression languages used. Instead, they are defined in terms of generic characteristics such as their LTS semantics. As a result, extending the monitor language with constructs such as parallel composition would not affect the existing framework. The instrumentation relation one adopts for composing monitors with systems necessarily affects the compositional properties and the correspondence between the respective definitions. However, these changes do not impinge on the general structure of our definitions and should be local to the detection condition in the resp. controllability definitions, namely Definition 11(i), 19(i) and 24(i), and the reachability-set definitions \( \text{aft}(\cdot), \text{saft}(\cdot) \) and \( \text{osaf}(\cdot) \) of Definitions 10, 18, and 24.

Future work. We will further investigate the implementability aspects of our analysis, possibly as an extension to existing model-checking tools. This may raise further issues and adjustments to our definitions e.g., it may be more efficient to batch the satisfiability checks in Definition 24. We plan to apply this to existing transition-based monitor specifications such as [5, 46] and validate its feasibility as an automated specification assistant.

Related Work. The need for determinising monitor syntheses from logical specifications is frequently discussed in the literature [7, 23]. In [1], the authors employ a trace-based definition of deterministic monitors that takes into consideration verdicts (similar to our definition for consistent detections of Definition 6 but without considering universal quantification over system instrumentations) and establish complexity bounds for determinising monitors wrt. this definition. Set-simulations, which are related to our monitor-sets, have been used as a proof technique for testing preorders in [15] but do not consider symbolic analyses. Acceptor ambiguity [29] is closely related to our notion of consistent detection with respect to the three outcomes of acceptance, rejection and withholding, as specified in Definition 6. Crucially, however, our definition universally quantifies over all possible system compositions. Subsequently, the main endeavour of our work was that of developing sound and complete compositional techniques to alleviate the analysis for consistent detection; we are unaware of any compositional or coinductive techniques used for determining acceptor ambiguity. Symbolic LTSs were studied extensively for value-passing CCS in [27, 28], but their use in controllability for reasoning about consistent monitor detections is, to our knowledge, novel. The particular setting where it is used, namely the instrumentation composition relation and the use of monitor-sets, also require new technical machinery, such as that of Definition 18. Our definition of controllability, Definition 11, is related to viability (usability) for clients in contract compliance [39] and must-testing [9]. Particularly, in the case of compliance, viability is defined coinductively and is satisfied whenever there exists a server that can engage with the client so as to lead it to success whenever interaction terminates. Apart from the universal quantifications over systems (viability existentially quantifies over servers), our work differs from [39, 9] wrt. the treatment of verdicts considered, the composition relation used (i.e., instrumentation), and the development of a symbolic analysis for handling of action/event data.

Acknowledgements. The author acknowledges the Dagstuhl seminar 17051 and would like to thank Luca Aceto, Antonis Achilleos, Duncan Attard, Giovanni Bernardi, Ian Cassar, Anna Ingólfsdóttir, Giles Reger and anonymous reviewers for their help and suggestions.


Flow Logic\textsuperscript{*†}

Orna Kupferman\textsuperscript{1} and Gal Vardi\textsuperscript{2}

\textsuperscript{1} School of Computer Science and Engineering, The Hebrew University, Israel
\textsuperscript{2} School of Computer Science and Engineering, The Hebrew University, Israel

\textbf{Abstract}

A flow network is a directed graph in which each edge has a capacity, bounding the amount of flow that can travel through it. Flow networks have attracted a lot of research in computer science. Indeed, many questions in numerous application areas can be reduced to questions about flow networks. This includes direct applications, namely a search for a maximal flow in networks, as well as less direct applications, like maximal matching or optimal scheduling. Many of these applications would benefit from a framework in which one can formally reason about properties of flow networks that go beyond their maximal flow.

We introduce \textit{Flow Logics}: modal logics that treat flow functions as explicit first-order objects and enable the specification of rich properties of flow networks. The syntax of our logic BFL\textsuperscript{*} (Branching Flow Logic) is similar to the syntax of the temporal logic CTL\textsuperscript{*}, except that atomic assertions may be flow propositions, like $> \gamma$ or $\geq \gamma$, for $\gamma \in \mathbb{N}$, which refer to the value of the flow in a vertex, and that first-order quantification can be applied both to paths and to flow functions. For example, the BFL\textsuperscript{*} formula $\exists((\geq 100) \land AG(low \rightarrow (\leq 20)))$ states that there is a legal flow function in which the flow is above 100 and in all paths, the amount of flow that travels through vertices with low security is at most 20.

We present an exhaustive study of the theoretical and practical aspects of BFL\textsuperscript{*}, as well as extensions and fragments of it. Our extensions include flow quantifications that range over non-integral flow functions or over maximal flow functions, path quantification that ranges over paths along which non-zero flow travels, past operators, and first-order quantification of flow values. We focus on the model-checking problem and show that it is PSPACE-complete, as it is for CTL\textsuperscript{*}. Handling of flow quantifiers, however, increases the complexity in terms of the network to P\textsuperscript{NP}, even for the LFL and BFL fragments, which are the flow-counterparts of LTL and CTL. We are still able to point to a useful fragment of BFL\textsuperscript{*} for which the model-checking problem can be solved in polynomial time.

1998 ACM Subject Classification F.4.1 Temporal Logic

Keywords and phrases Flow Network, Temporal Logic

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2017.9

\section{Introduction}

A \textit{flow network} is a directed graph in which each edge has a capacity, bounding the amount of flow that can travel through it. The amount of flow that enters a vertex equals the amount of flow that leaves it, unless the vertex is a \textit{source}, which has only outgoing flow, or a \textit{target}, which has only incoming flow. The fundamental \textit{maximum-flow problem} gets as input a flow

\textsuperscript{*} This research has received funding from the European Research Council under the EU’s 7-th Framework Programme (FP7/2007-2013) / ERC grant agreement no 278410.

\textsuperscript{†} A full version of the paper is available at \url{http://www.cs.huji.ac.il/~ornak/publications/concur17.pdf}.
network and searches for a maximal flow from the source to the target [10, 18]. The problem was first formulated and solved in the 1950’s [16, 17]. It has attracted much research on improved algorithms [12, 11, 19, 32] and applications [1].

The maximum-flow problem can be applied in many settings in which something travels along a network. This covers numerous application domains, including traffic in road or rail systems, fluids in pipes, currents in an electrical circuit, packets in a communication network, and many more [1]. Less obvious applications involve flow networks that are constructed in order to model settings with an abstract network, as in the case of scheduling with constraints [1] or elimination in partially completed tournaments [36]. In addition, several classical graph-theory problems can be reduced to the maximum-flow problem. This includes the problem of finding a maximum bipartite matching, minimum path cover, maximum edge-disjoint or vertex-disjoint path, and many more [10, 1]. Variants of the maximum-flow problem can accommodate further settings, like circulation problems [38], multiple source and target vertices, costs for unit flows, multiple commodities, and more [14].

All the above applications reduce the problem at hand to the problem of finding a maximal flow in a network. Often, however, one would like to reason about properties of flow networks that go beyond their maximal flow. This is especially true when the vertices or edges of the network attain information to which the properties can refer. For example, the vertices of a network may be labeled by their security level, and we may want to check whether all legal flow functions are such that the flow in every low-security vertex is at most 20, or check whether there is a flow function in which more than 100 units of flow reach the target and still the flow in every low-security vertex is at most 20. As another example, assume that each vertex in the network is labeled by the service provider that owns it, and we want to find a maximal flow under the constraint that flow travels through vertices owned by at most two providers.

The challenge of reasoning about properties of systems has been extensively studied in the context of formal verification. In particular, in temporal-logic model checking [8, 34], we check whether a system has a desired property by translating the system into a labeled state-transition graph, translating the property into a temporal-logic formula, and deciding whether the graph satisfies the formula. Model checking is one of the notable success stories of theoretical computer science, with exciting theoretical research that is being transformed into industrial applications [9, 7]. By viewing networks as labeled state-transition graphs, we can use existing model-checking algorithms and tools in order to reason about the structural properties of networks. We can check, for example, that every path from the source to the target eventually visits a check-sum vertex. Most interesting properties of flow networks, however, refer to flows and their values, and not just to the structural properties of the network. Traditional temporal logics do not support the specification and verification of such properties.

We introduce and study Flow Logics: modal logics that treat flow functions as explicit first-order objects and enable the specification of rich properties of flow networks. The syntax of our logic BFL* (Branching Flow Logic) is similar to the syntax of the temporal logic CTL*, except that atomic assertions are built from both atomic propositions and flow propositions, like $\gamma > \gamma$ or $\gamma \geq \gamma$, for $\gamma \in \mathbb{N}$, which refer to the value of the flow in a vertex, and that first-order quantification can be applied both to paths and to flow functions. Thus, in addition to the path quantifiers $A$ (“for all paths”) and $E$ (“there exists a path”) that range over paths, states formulas may contain the flow quantifiers $A$ (“for all flow functions”) and $E$ (“there exists a flow function”). For example, the BFL* formula $E((\geq 100) \land AG(low \rightarrow (\leq 20)))$ states the property discussed above, namely that there is a flow function in which the value
of the flow is at least 100, and in all paths, the value of flow in vertices with low security is at most 20.

We study the theoretical aspects of BFL* as well as extensions and fragments of it. We demonstrate their applications in reasoning about flow networks, and we examine the complexity of their model-checking problem. Below we briefly survey our results. (1) We show that while maximal flow can always be achieved by integral flows [16], in the richer setting of flow logic, restricting attention to integral flows may change the satisfaction value of formulas. Accordingly, our semantics for BFL* considers two types of flow quantification: one over integral flows and another over non-integral ones. (2) We prove that bisimulation [33] is not a suitable equivalence relation for flow logics, which are sensitive to unwinding. We relate this to the usefulness of past operators in flow logic, and we study additional aspects of the expressive power of BFL*. (3) We consider extensions of BFL* by path quantifiers that range over paths on which flow travels (rather than over all paths in the network), and by first-order quantification on flow values. (4) We study the model-checking complexity of BFL*, its extensions, and some natural fragments. We show that algorithms for temporal-logic model-checking can be extended to handle flow logics, and that the complexity the BFL* model-checking problems is PSPACE-complete. We study also the network complexity of the problem, namely the complexity in terms of the network, assuming that the formula is fixed [29, 27], and point to a fragment of BFL* for which the model-checking problem can be solved in polynomial time.

Related Work. There are three types of related works: (1) efforts to generalize the maximal-flow problem to richer settings, (2) extensions of temporal logics by new elements, in particular first-order quantification over new types, and (3) works on logical aspects of networks and their use in formal methods. Below we briefly survey them and their relation to our work.

As discussed early in this section, numerous extensions to the classical maximal-flow problems have been considered. In particular, some works that add constraints on the maximal flow, like capacities on vertices, or lower bounds on the flow along edges. Closest to flow logics are works that refer to labeled flow networks. For example, [20] considers flow networks in which edges are labeled, and the problem of finding a maximal flow with a minimum number of labels. Then, the maximal utilization problem of capacitated automata [26] amounts to finding maximal flow in a labeled flow network where flow is constrained to travel only along paths that belong to a given regular language. Our work suggests a formalism that embodies all these extensions, as well as a framework for formally reasoning about many more extensions and settings.

The competence of temporal-logic model checking initiated numerous extensions of temporal logics, aiming to capture richer settings. For example, real-time temporal logics include clocks with a real-time domain [2], epistemic temporal logics include knowledge operators [21], and alternating temporal logics include game modalities [3]. Closest to our work is strategy logic [6], where temporal logic is enriched by first-order quantification of strategies in a game. Beyond the theoretical interest in strategy logic, it was proven useful in synthesizing strategies in multi-agent systems and in the solution of rational synthesis [15].

Finally, network verification is an increasingly important topic in the context of protocol verification [23]. Tools that allow verifying properties of network protocols have been developed [40, 31]. These tools support verification of network protocols in the design phase as well as runtime verification [22]. Some of these tools use a query language called Network Datalog in order to specify network protocols [30]. Verification of Software Defined Net-
works has been studied widely, for example in [24, 28, 5]. Verification of safety properties in networks with finite-state middleboxes was studied in [39]. Network protocols describe forwarding policies for packets, and are thus related to specific flow functions. However, the way traffic is transmitted in these protocols does not correspond to the way flow travels in a flow network. Thus, properties verified in this line of work are different from these we can reason about with flow logic.

Due to the lack of space, some details and proofs are omitted, and can be found in the full version, in the authors’ URLs.

2 The Flow Logic BFL*

A flow network is \( N = \langle AP, V, E, c, \rho, s, T \rangle \), where \( AP \) is a set of atomic propositions, \( V \) is a set of vertices, \( s \in V \) is a source vertex, \( T \subseteq V \) is a set of target vertices, \( E \subseteq (V \setminus T) \times (V \setminus \{s\}) \) is a set of directed edges, \( c : E \to \mathbb{N} \) is a capacity function, assigning to each edge an integral amount of flow that the edge can transfer, and \( \rho : V \to 2^{AP} \) assigns each vertex \( v \in V \) to the set of atomic propositions that are valid in \( v \). Note that no edge enters the source vertex or leaves a target vertex. We assume that all vertices \( t \in T \) are reachable from \( s \) and that each vertex has at least one target vertex reachable from it. For a vertex \( u \in V \), let \( E^u \) and \( E_u \) be the sets of incoming and outgoing edges to and from \( u \), respectively. That is, \( E^u = (V \times \{u\}) \cap E \) and \( E_u = (\{u\} \times V) \cap E \).

A flow is a function \( f : E \to \mathbb{N} \) that describes how flow is directed in \( N \). The capacity of an edge bounds the flow in it, thus for every edge \( e \in E \), we have \( f(e) \leq c(e) \). All incoming flow must exit a vertex, thus for every vertex \( v \in V \setminus (\{s\} \cup T) \), we have \( \sum_{e \in E_v} f(e) = \sum_{e \in E_v} c(e) \). We extend \( f \) to vertices and use \( f(v) \) to denote the flow that travels through \( v \). Thus, for \( v \in V \setminus (\{s\} \cup T) \), we define \( f(v) = \sum_{e \in E_v} f(e) \), for the source vertex \( s \), we define \( f(s) = \sum_{e \in E_s} f(e) \), and for a target vertex \( t \in T \), we define \( f(t) = \sum_{e \in E_t} f(e) \). Note that the preservation of flow in the internal vertices guarantees that \( f(s) = \sum_{t \in T} f(t) \), which is the amount of flow that travels from \( s \) to all the target vertices together. We say that a flow function \( f \) is maximal if for every flow function \( f' \), we have \( f'(s) \leq f(s) \). A maximal flow function can be found in polynomial time [17]. The maximal flow for \( N \) is then \( f(s) \) for some maximal flow function \( f \).

The logic BFL* is a Branching Flow Logic that can specify properties of networks and flows in them. As in CTL*, there are two types of formulas in BFL*: state formulas, which describe vertices in a network, and path formulas, which describe paths. In addition to the operators in CTL*, the logic BFL* has flow propositions, with which one can specify the flow in vertices, and flow quantifiers, with which one can quantify flow functions universally or existentially. When flow is not quantified, satisfaction is defined with respect to both a network and a flow function. Formally, given a set \( AP \) of atomic propositions, a BFL* state formula is one of the following:

(S1) An atomic proposition \( p \in AP \).
(S2) A flow proposition \( > \gamma \) or \( \geq \gamma \), for an integer \( \gamma \in \mathbb{N} \).
(S3) \( \neg \varphi \) or \( \varphi_1 \lor \varphi_2 \), for BFL* state formulas \( \varphi_1 \) and \( \varphi_2 \).
(S4) \( A\psi \), for a BFL* path formula \( \psi \). \( A \) is a path quantifier.
(S5) \( A\varphi \), for a BFL* state formula \( \varphi \). \( A \) is a flow quantifier.

A BFL* path formula is one of the following:

(P1) A BFL* state formula.
(P2) \( \neg \psi_1 \) or \( \psi_1 \lor \psi_2 \), for BFL* path formulas \( \psi_1 \) and \( \psi_2 \).
(P3) \( X\psi_1 \) or \( \psi_1 U\psi_2 \), for BFL* path formulas \( \psi_1 \) and \( \psi_2 \).
We say that a BFL* formula \( \varphi \) is closed if all flow propositions appear in the scope of a flow quantifier. The logic BFL* consists of the set of closed BFL* state formulas. We refer to state formula of the form \( A\varphi \) as a flow state formula.

The semantics of BFL* is defined with respect to vertices in a flow network. Before defining the semantics, we need some more definitions and notations. Let \( N = (AP, V, E, c, \rho, s, T) \). For two vertices \( u \) and \( w \) in \( V \), a finite sequence \( \pi = v_0, v_1, \ldots, v_k \in V^* \) of vertices is a \((u, w)\)-path in \( N \) if \( v_0 = u, v_k = w \), and \((v_i, v_{i+1}) \in E \) for all \( 0 \leq i < k \). If \( w \in T \), then \( \pi \) is a target \( u \)-path.

State formulas are interpreted with respect to a vertex \( v \) in \( N \) and a flow function \( f : E \to \mathbb{N} \). When the formula is closed, satisfaction is independent of the function \( f \) and we omit it. We use \( v, f \models \varphi \) to indicate that the vertex \( v \) satisfies the state formula \( \varphi \) when the flow function is \( f \). The relation \( \models \) is defined inductively as follows.

\[(S1) \text{ For an atomic proposition } p \in AP, \text{ we have that } v, f \models p \text{ iff } p \in \rho(v).\]
\[(S2) \text{ For } \gamma \in \mathbb{N}, \text{ we have } v, f \models \varphi \text{ iff } f(v) > \gamma \text{ and } v, f \models \varphi \text{ iff } f(v) \geq \gamma.\]
\[(S3a) v, f \models \neg \varphi_1 \text{ iff } v, f \not\models \varphi_1.\]
\[(S3b) v, f \models \varphi_1 \lor \varphi_2 \text{ iff } v, f \models \varphi_1 \text{ or } v, f \models \varphi_2.\]
\[(S4) v, f \models A\psi \text{ iff for all target } v\text{-paths } \pi, \text{ we have that } \pi, f \models \psi.\]
\[(S5) v, f \models A\varphi \text{ iff for all flow functions } f', \text{ we have } v, f' \models \varphi.\]

Path formulas are interpreted with respect to a finite path \( \pi \) in \( N \) and a flow function \( f : E \to \mathbb{N} \). We use \( \pi, f \models \varphi \) to indicate that the path \( \pi \) satisfies the path-flow formula \( \psi \) when the flow function is \( f \). The relation \( \models \) is defined inductively as follows. Let \( \pi = v_0, v_1, \ldots, v_k \). For \( 0 \leq i \leq k \), we use \( \pi^i \) to denote the suffix of \( \pi \) that starts at \( v_i \), thus \( \pi^i = v_i, v_{i+1}, \ldots, v_k \).

\[(P1) \text{ For a state formula } \varphi, \text{ we have that } \pi, f \models \varphi \text{ iff } v_0, f \models \varphi.\]
\[(P2a) \pi, f \models \neg \psi \text{ iff } \pi, f \not\models \psi.\]
\[(P2b) \pi, f \models \psi_1 \lor \psi_2 \text{ iff } \pi, f \models \psi_1 \text{ or } \pi, f \models \psi_2.\]
\[(P3a) \pi, f \models X\psi_1 \text{ iff } k > 0 \text{ and } \pi^i, f \models \psi_1.\]
\[(P3b) \pi, f \models \psi_1 U\psi_2 \text{ iff there is } j \leq k \text{ such that } \pi^j, f \models \psi_2, \text{ and for all } 0 \leq i < j, \text{ we have } \pi^i, f \models \psi_1.\]

For a network \( N \) and a closed BFL* formula \( \varphi \), we say that \( N \) satisfies \( \varphi \), denoted \( N \models \varphi \), iff \( s \models \varphi \) (note that since \( \varphi \) is closed, we do not specify a flow function).

Additional Boolean connectives and modal operators are defined from \( \neg, \lor, X, \text{ and } U \) in the usual manner; in particular, \( F\psi = \text{true} U\psi \) and \( G\psi = \neg F\neg\psi \). We also define dual and abbreviated flow propositions: \( < \gamma = \neg(\geq \gamma), \leq \gamma = \neg(> \gamma), \text{ and } \gamma = (\leq \gamma) \land (\geq \gamma), \text{ a dual path quantifier: } E\psi = \neg A\neg\psi, \text{ and a dual flow quantifier: } E\varphi = \neg A\varphi.\)

**Example 1.** Consider a network \( N \) in which target vertices are labeled by an atomic proposition \textit{target}, and low-security vertices are labeled \textit{red}. The BFL* formula \( EEF(\text{target} \land 20) \) states that there is a flow in which \( 20 \) units reach a target vertex, and the BFL* formula \( A(\geq 20) \rightarrow AX(\geq 4) \) states that in all flow functions in which the flow at the source is at least \( 20 \), all the successors must have flow of at least \( 4 \). Finally, \( E(\geq 100) \land AG(\text{red} \rightarrow (\leq 20)) \) states that there is a flow of at least \( 100 \) in which the flow in every low-security vertex is at most \( 20 \), whereas \( A(> 200) \rightarrow E\text{\textit{red}} \land (\geq 200) \) states that when the flow is above \( 200 \), then there must exist a low-security vertex in which the flow is above \( 20 \). As an example to a BFL* formula with an alternating nesting of flow quantifiers, consider the formula \( EAG(< 10 \rightarrow A < 15) \), stating that there is a flow such that wherever the flow is below \( 10 \), then in every flow it would be below \( 15 \).
Remark. Note that while the semantics of CTL\(^*\) and LTL is defined with respect to infinite trees and paths, path quantification in BFL\(^*\) ranges over finite paths. We are still going to use techniques and results known for CTL\(^*\) and LTL in our study. Indeed, for upper bounds, the transition to finite computations only makes the setting simpler. Also, lower-bound proofs for CTL\(^*\) and LTL are based on an encoding of finite runs of Turing machines, and apply also to finite paths.

Specifying finite paths, we have a choice between weak and strong semantics for the \(X\) operator. In the weak semantics, the last vertex in a path satisfies \(X\psi\), for all \(\psi\). In particular, it is the only vertex that satisfies \(X\false\). In the strong semantics, the last vertex does not satisfy \(X\psi\), for all \(\psi\). In particular, it does not satisfy \(X\true\). We use the strong semantics.

3 Properties of BFL\(^*\)

3.1 Integral vs. non-integral flow functions

Our semantics of BFL\(^*\) considers integral flow functions: vertices receive integral incoming flow and partition it to integral flows in the outgoing edges. Integral-flow functions arise naturally in settings in which the objects we transfer along the network cannot be partitioned into fractions, as is the case with cars, packets, and more. Sometimes, however, as in the case of liquids, flow can be partitioned arbitrarily. In the traditional maximum-flow problem, it is well known that the maximum flow can be achieved by integral flows [16]. We show that, interestingly, in the richer setting of flow logic, restricting attention to integral flows may change the satisfaction value of formulas.

\[\text{Proposition 2.} \]

Allowing the quantified flow functions in BFL\(^*\) to get values in \(\mathbb{R}\) changes its semantics.

\[\text{Proof.} \]

Consider the network on the right. The BFL\(^*\) formula \(\varphi = E(1 \land AX(> 0))\) states that there is a flow function in which the flow that leaves the source is 1 and the flow of both its successors is strictly positive. It is easy to see that while no integral flow function satisfies the requirement in \(\varphi\), a flow function in which 1 unit of flow in \(s\) is partitioned between \(u\) and \(v\) does satisfy it.

Proposition 2 suggests that quantification of flow functions that allow non-integral flows may be of interest. In Section 4.3 we discuss such an extension.

3.2 Sensitivity to unwinding

For a network \(N = (AP, V, E, c, \rho, s, T)\), let \(N_t\) be the unwinding of \(N\) into a tree. Formally, \(N_t = (AP, V', E', \rho', s, T')\), where \(V' \subseteq V^*\) is the smallest set such that \(s \in V'\), and for all \(w \cdot v \in V'\) with \(w \in V^*\) and \(v \in V \setminus T\), and all \(u \in V\) such that \(E(v, u)\), we have that \(w \cdot v \cdot u \in V'\), with \(\rho'(w \cdot v \cdot u) = \rho(u)\). Also, \((w \cdot v, w \cdot v \cdot u) \in E'\), with \(c'((w \cdot v, w \cdot v \cdot u)) = c((v, u))\). Finally, \(T' = V' \cap (V^* \setminus T)\). Note that \(N_t\) may be infinite. Indeed, a cycle in \(N\) induces infinitely many vertices in \(N_t\).

The temporal logic CTL\(^*\) is insensitive to unwinding. Indeed, \(N\) and \(N_t\) are bisimilar, and for every CTL\(^*\) formula \(\varphi\), we have \(N \models \varphi\) if \(N_t \models \varphi\) [33]. We show that this is not the case for BFL\(^*\).

\[\text{Proposition 3.} \]

The value of the maximal flow is sensitive to unwinding.
Proof. Consider the network $N$ appearing in Figure 1, and its unwinding $N_t$ which appears in its right. It is easy to see that the value of the maximal flow in $N$ is 7 and the value of the maximal flow from $s$ to $T'$ in $N_t$ is 8.

\[\begin{figure}
\begin{center}
\begin{tabular}{c}
\includegraphics[width=0.5\textwidth]{figure1}
\end{tabular}
\end{center}
\caption{The flow network $N$ and its unwinding $N_t$.}
\end{figure}\]

\[\begin{figure}
\begin{center}
\begin{tabular}{c}
\includegraphics[width=0.5\textwidth]{figure2}
\end{tabular}
\end{center}
\caption{Assigning workers to jobs.}
\end{figure}\]

\[\begin{figure}
\begin{center}
\begin{tabular}{c}
\includegraphics[width=0.5\textwidth]{figure3}
\end{tabular}
\end{center}
\caption{Positive path quantification.}
\end{figure}\]

\[\begin{align}
\text{Figure 1} & \text{ The flow network } N \text{ and its unwinding } N_t. \\
\text{Figure 2} & \text{ Assigning workers to jobs.} \\
\text{Figure 3} & \text{ Positive path quantification.}
\end{align}\]

\[\begin{align}
\text{Proof.} & \text{ Consider the network } N \text{ appearing in Figure 1, and its unwinding } N_t \text{ which appears in its right. It is easy to see that the value of the maximal flow in } N \text{ is 7 and the value of the maximal flow from } s \text{ to } T' \text{ in } N_t \text{ is 8.} \\
\text{Corollary 4.} & \text{ The logic } BFL^* \text{ is sensitive to unwinding.}
\end{align}\]

The sensitivity of $BFL^*$ to unwinding suggests that extending $BFL^*$ with past operators can increase its expressive power. In Section 4.4, we discuss such an extension.

\section{Extensions and Fragments of $BFL^*$}

In this section we discuss useful extensions and variants of $BFL^*$, as well as fragments of it. As we shall show in the sequel, while the extensions come with no computational price, their model checking requires additional techniques.

\subsection{Positive path quantification}

Consider a network $N = \langle AP, V, E, c, \rho, s, T \rangle$ and a flow function $f : E \to \mathbb{N}$. We say that a path $\pi = v_0, v_1, \ldots, v_k$ is positive if the flow along all the edges in $\pi$ is positive. Formally, $f(v_i, v_{i+1}) > 0$, for all $0 \leq i < k$. Note that it may be that $f(v_i) > 0$ for all $0 \leq i < k$ and still $\pi$ is not positive. It is sometimes desirable to restrict the range of path quantification to paths along which flow travels. This is the task of the positive path quantifier $A^+$, with the following semantics (dually, $E^+ \psi = \neg A^+ \neg \psi$).

\[\begin{align}
v, f & |\Rightarrow A^+ \psi \text{ iff for all positive target } v\text{-paths } \pi, \text{ we have that } \pi, f |\Rightarrow \psi.
\end{align}\]

\[\begin{align}
\text{Example 5.} & \text{ Let } W \text{ be a set of workers and } J \text{ be a set of jobs. Each worker } w \in W \text{ can be assigned to perform jobs from a subset } J_w \subseteq J. \text{ It is required to perform all jobs by assigning exactly one worker to each job and at most two jobs to each worker. This problem can be solved using a flow network. Indeed, a flow of } k \text{ units in the network described in Figure 2 corresponds to a legal assignment in which } k \text{ jobs are performed (in the figure,}
\end{align}\]
edges with no specified capacity have capacity 1). Now assume that some jobs should be processed in Location $a$ and the others in Location $b$. A worker can process jobs only in a single location, $a$ or $b$. By labeling the job-vertices by their location, the existence of a legal assignment in which $k$ jobs are processed can be expressed by the BFL$^*$ formula $E(k \land AX(A^+Xa \lor A^+Xb))$, which uses positive path quantification.

### 4.2 Maximal flow quantification

It is sometimes desirable to restrict the range of flow quantification to maximal flow functions. This is the task of the maximal-flow quantifier $A^\text{max}$, with the following semantics (dually, $E^{\text{max}}\varphi = \neg A^{\text{max}}\neg \varphi$).

- $v, f \models A^{\text{max}}\varphi$ iff for all maximal-flow functions $f'$, we have that $v, f' \models \varphi$.

In a similar manner, it is sometimes helpful to relate to the maximal flow in the network. The max-flow constant $\gamma_{\text{max}} \in \mathbb{N}$ maintains the value of the maximal flow from $s$ to $T$. We also allow arithmetic operations on $\gamma_{\text{max}}$.

#### Example 6.
Recall the job-assignment problem from Example 5. The formula $E^{\text{max}}(AX(A^+Xa \lor A^+Xb))$ states that the requirements about the locations do not reduce the number of jobs assigned without this requirement. Then, the formula $E((\geq \gamma_{\text{max}} - 4) \land AX(A^+Xa \lor A^+Xb))$ states that the requirements about the locations may reduce the number of jobs performed by at most 4.

### 4.3 Non-integral flow quantification

As discussed in Section 3.1, letting flow quantification range over non-integral flow functions may change the satisfaction value of a BFL$^*$ formula. We extend BFL$^*$ with a non-integral flow quantifier $A^\text{IR}$, with the following semantics (dually, $E^{\text{IR}}\varphi = \neg A^{\text{IR}}\neg \varphi$).

- $v, f \models A^{\text{IR}}\varphi$ iff for all flow functions $f' : E \rightarrow \mathbb{R}$, we have that $v, f' \models \varphi$.

### 4.4 Past operators

As discussed in Section 3.2, while temporal logics are insensitive to unwinding, this is not the case for BFL$^*$. Intuitively, this follows from the fact that the flow in a vertex depends on the flow it gets from all its predecessors. This dependency suggests that an explicit reference to predecessors is useful, and motivates the extension of BFL$^*$ by past operators.

Adding past to a branching logic, one can choose between a linear-past semantics – one in which past is unique (technically, the semantics is with respect to an unwinding of the network), and a branching-past semantics – one in which all the possible behaviors that lead to present are taken into an account (technically, the semantics is dual to that of future operators, and is defined with respect to the network) [25]. For flow logics, the branching-past approach is the suitable one, and is defined as follows.

For a path $\pi = v_0, v_1, \ldots, v_k \in V^*$, a vertex $v \in V$, and index $0 \leq i \leq k$, we say that $\pi$ is a source-target $(v, i)$-path if $v_0 = s$, $v_i = v$, and $v_k \in T$. We add to BFL$^*$ two past modal operators, $Y$ (“Yesterday”) and $S$ (“Since”), and adjust the semantics as follows. Defining the semantics of logics that refer to the past, the semantics of path formulas is defined with respect to a path and an index in it. We use $\pi, i, f \models \psi$ to indicate that the path $\pi$ satisfies the path formula $\psi$ from position $i$ when the flow function is $f$. For state formulas, we adjust the semantics as follows.

$\text{(S4)}$ $v, f \models A\psi$ iff for all source-target $(v, i)$-paths $\pi$, we have that $\pi, i, f \models \psi$. 
Then, for path formulas, we have the following (the adjustment to refer to the index \( i \) in all other modalities is similar).
\[
\begin{align*}
\pi, i, f \models \gamma \psi_1 \iff i > 0 \text{ and } \pi, i - 1, f \models \psi_1.
\pi, i, f \models \psi_1 \psi_2 \iff \text{there is } 0 \leq j < i \text{ such that } \pi, j, f \models \psi_2, \text{ and for all } j + 1 \leq l \leq i, \text{ we have } \pi, l, f \models \psi_1.
\end{align*}
\]

\textbf{Example 7.} Recall the job-assignment problem from Example 5. Assume we want to apply the restriction about the location only to jobs that can be assigned only to workers with no car. Thus, if all the predecessors of a job-vertex are labeled by \textit{car}, then this job can be served in either locations. Using past operators, we can specify this property by
\[
E(k \wedge AX(A^+ X(a \lor AY \textit{car}) \lor A^+ X(b \lor AY \textit{car}))).
\]

As proven in [25], adding past to CTL* with a branching-past semantics strictly increases its expressive power. The same arguments can be used in order to show that BFL* with past operators is strictly more expressive than BFL*.

4.5 First-Order quantification on flow values

The flow propositions in BFL* include constants. This makes it impossible to relate the flow in different vertices other than specifying all possible constants that satisfy the relation. In BFL* \textit{with quantified flow values} we add flow variables \( X = \{x_1, \ldots, x_n\} \) that can be quantified universally or existentially and specify such relations conveniently. We also allow the logic to apply arithmetic operations on the values of variables in \( X \).

For a set of arithmetic operators \( O \) (that is, \( O \) may include \(+, \ast, \text{ etc.} \), let BFL*(\( O \)) be BFL* in which Rule S2 is extended to allow expressions with variables in \( X \) constructed by operators in \( O \), and we also allow quantification on the variables in \( X \). Formally, we have the following:

\textbf{(S2)} A flow proposition \( \geq g(x_1, \ldots, x_k) \) or \( \geq g(x_1, \ldots, x_k) \), where \( x_1, \ldots, x_k \) are variables in \( X \) and \( g \) is an expression obtained from \( x_1, \ldots, x_k \) by applying operators in \( O \), possibly using constants in \( \mathbb{N} \). We assume that \( g : \mathbb{N}^k \rightarrow \mathbb{N} \). That is, \( g \) leaves us in the domain \( \mathbb{N} \).

\textbf{(S6)} \( \forall x \varphi \), for \( x \in X \) and a BFL*(\( O \)) formula \( \varphi \) in which \( x \) is free.

For a BFL*(\( O \)) formula \( \varphi \) in which \( x \) is a free variable, and a constant \( \gamma \in \mathbb{N} \), let \( \varphi[x \leftarrow \gamma] \) be the formula obtained by assigning \( \gamma \) to \( x \) and replacing expressions by their evaluation. Then, \( v, f \models \forall x \varphi \) iff for all \( \gamma \in \mathbb{N} \), we have that \( v, f \models \varphi[x \leftarrow \gamma] \).

\textbf{Example 8.} The logic BFL*(\( \emptyset \)) includes the formula \( \mathcal{E}AG \forall x(\text{split} \wedge x > 0) \rightarrow EX(> 0 \wedge < x) \), stating that there is a flow in which all vertices that are labeled \textit{split} and with a positive flow \( x \) have a successor in which the flow is positive but strictly smaller than \( x \). Then, BFL*(\( \text{div} \)) includes the formula \( \mathcal{E}AG \forall x(x \rightarrow EX(\geq x \text{ div } 2)) \), stating that there is a flow in which all vertices have a successor that has at least half of their flow.

Finally, BFL*(\( + \)) includes the formula \( \exists x \exists y \mathcal{E}^{\text{max}} AG(\neg(\text{source} \lor \text{target}) \rightarrow x \lor y \lor (x+y)) \), stating that there are values \( x \) and \( y \), such that it is possible to attain the maximal flow by assigning to all vertices, except maybe source and target vertices, values in \( \{x, y, x+y\} \).

\footnote{We note that the result from [25] does not immediately imply the addition of expressive power, as it is based on the fact that only CTL* with past operators is sensitive to unwinding (and, as we prove in Section 3.2, BFL* is sensitive to unwinding). Still, since the specific formula used in [25] in order to prove the sensitivity of CTL* with past to unwinding does not refer to flow, it is easy to see that it has no equivalent BFL* formula.}
4.6 Fragments of BFL*

For the temporal logic CTL*, researchers have studied several fragments, most notably LTL and CTL. In this section we define interesting fragments of BFL*.

Flow-CTL* and Flow-LTL

The logics Flow-CTL* and Flow-LTL are extensions of CTL* and LTL in which atomic state formulas may be, in addition to APs, also the flow propositions $>\gamma$ or $\geq \gamma$, for an integer $\gamma \in \mathbb{N}$. Thus, no quantification on flow is allowed, but atomic formulas may refer to flow. The semantics of Flow-CTL* is defined with respect to a network and a flow function, and that of Flow-LTL is defined with respect to a path in a network and a flow function.

Linear Flow Logic

The logic LFL is the fragment of BFL* in which only one external universal path quantification is allowed. Thus, an LFL formula is a BFL* formula of the form $A\psi$, where $\psi$ is generated without rule S4.

Note that while the temporal logic LTL is a “pure linear” logic, in the sense that satisfaction of an LTL formula in a computation of a system is independent of the structure of the system, the semantics of LFL mixes linear and branching semantics. Indeed, while all the paths in $N$ have to satisfy $\psi$, the context of the system is important. To see this, consider the LFL formula $\varphi = A((\geq 10) \rightarrow X(\geq 4))$. The formula states that in all paths, all flow functions in which the flow at the first vertex in the path is at least 10, are such that the flow at the second vertex in the path is at least 4. In order to evaluate the path formula $A((\geq 10) \rightarrow X(\geq 4))$ in a path $\pi$ of a network $N$ we need to know the capacity of all the edges from the source of $N$, and not only the capacity of the first edge in $\pi$. For example, $\varphi$ is satisfied in networks in which there are two successors to the source, each connected by an edge with capacity 4, 5, or 6. Consider now the LFL formula $\varphi' = A\epsilon(10 \land X(\geq 4))$. Note that $\varphi'$ is not equal to the BFL* formula $\theta = A\epsilon(10 \land AX(\geq 4))$. Indeed, in the latter, the same flow function should satisfy the path formula $X(\geq 4)$ in all paths.

No nesting of flow quantifiers

The logic $BFL_1^*$ contains formulas that are Boolean combinations of formulas of the form $E\varphi$ and $A\varphi$, for a Flow-CTL* formula $\varphi$. Of special interest are the following fragments of $BFL_1^*$:

- $\exists BFL_1^*$ and $\forall BFL_1^*$, where formulas are of the form $E\varphi$ and $A\varphi$, respectively, for a Flow-CTL* formula $\varphi$.
- $\exists LFL_1$ and $\forall LFL_1$, where formulas are of the form $E A \psi$ and $A A \psi$, respectively, for a Flow-LTL formula $\psi$, and LFL, where a formula is a Boolean combination of $\exists LFL_1$ and $\forall LFL_1$ formulas.

Conjunctive-BFL*

The fragment Conjunctive-BFL* (CBFL*, for short) contains BFL* formulas whose flow state sub-formulas restrict the quantified flow in a conjunctive way. That is, when we “prune” a CBFL* formula into requirements on the network, atomic flow propositions are only conjunctively related. This would have a computational significance in solving the model-checking problem.
Consider an \( \exists \text{BFL}_1^* \) formula \( \varphi = E \theta \). We say that an operator \( g \in \{ \vee, \wedge, E, A, X, F, G, U \} \) has a positive polarity in \( \varphi \) if all the occurrences of \( g \) in \( \theta \) are in a scope of an even number of negations. Dually, \( g \) has a negative polarity in \( \varphi \) if all its occurrences in \( \theta \) are in a scope of an odd number of negations.

The logic \( \exists \text{CBFL}_1^* \) is a fragment of \( \exists \text{BFL}_1^* \) in which the only operators with a positive polarity are \( \wedge, A, X, \) and \( G \), and the only operators with a negative polarity are \( \vee, E, X, \) and \( F \). Note that \( U \) is not allowed, as its semantics involves both conjunctions and disjunctions. Note that by pushing negations inside, we make all operators of a positive polarity outside so that path formulas do not have internal conjunctions – for example, \( AXAX\xi \) is nested inside another universal path quantification – for example, transform formulas that have \( \wedge \) polarity are path formulas in BFL.

Then, since all requirements are universal and conjunctively related, we can push conjunctions outside so that path formulas do not have internal conjunctions – for example, transform \( AX(\xi_1 \wedge \xi_2) \) into \( AX\xi_1 \land AX\xi_2 \), and can get rid of universal path quantification that is nested inside another universal path quantification – for example, transform \( AXAX\xi \) into \( AXX\xi \). Finally, since we use the strong semantics to \( X \) and \( G \), we can replace formulas that have \( X \) nested inside \( G \) by \text{false}.

The logic \( \forall \text{CBFL}_1^* \) is the dual fragment of \( \forall \text{BFL}_1^* \). In other words, \( A\theta \) is in \( \forall \text{CBFL}_1^* \) iff \( \neg E\theta \) is in \( \exists \text{CBFL}_1^* \). The logic \( \text{CBFL}_1^* \) is then obtained by going up a hierarchy in which formulas of lower levels serve as atomic propositions in higher levels.

We now define the syntax of \( \text{CBFL}_1^* \) formally. For simplicity, we define it in a normal form, obtained by applying the rules described above. A \( \text{CBFL}_1^* \) formula is a Boolean assertion over \( AP \). For \( i \geq 0 \), a \( \text{CBFL}_1^{i+1} \) formula is a Boolean assertion over \( \text{CBFL}_1^i \) formulas and formulas of the form \( E(\psi_1 \land \cdots \land \psi_n) \), where \( \psi_j \) is of the form \( X^{k_j} \xi_j \) or \( X^{k_j}G\xi_j \), where \( k_j \geq 0 \) and \( \xi_j \) is a \( \text{CBFL}_1^i \) formula or a flow proposition (that is, \( > \gamma, < \gamma, \geq \gamma, \) or \( \leq \gamma \), for an integer \( \gamma \in \mathbb{N} \)). Then, a \( \text{CBFL}_1^i \) formula is a \( \exists \text{CBFL}_1^i \) formula for some \( i \geq 0 \). Note that both \( \exists \text{CBFL}_1^i \) and \( \forall \text{CBFL}_1^i \) are contained in \( \text{CBFL}_1^i \).

Example 9. Recall the job-assignment problem from Example 5. The \( \text{CBFL}_1^i \) formula \( \mathcal{A}(< 10 \rightarrow EX \leq 0) \land E(15 \land AX \geq 1) \) states that if less than 10 jobs are processed, then at least one worker is unemployed, but it is possible to process 15 jobs and let every worker process at least one job.

BFL

The logic BFL is the fragment of BFL in which every modal operator \( (X, U) \) is preceded by a path quantifier. That is, it is the flow counterpart of CTL.

5 Model Checking

In this section we study the model-checking problem for BFL*. The problem is decide, given a flow network \( N \) and a BFL* formula \( \varphi \), whether \( N \models \varphi \).

Theorem 10. BFL* model checking is PSPACE-complete.

Proof. Consider a network \( N \) and a BFL* formula \( \varphi \). The idea behind our model-checking procedure is similar to the one that recursively employs LTL model checking in the process of CTL* model checking [13]. Here, however, the setting is more complicated. Indeed, the path formulas in BFL* are not “purely linear”, as the flow quantification in them refers to flow in the (branching) network. In addition, while the search for witness paths is restricted to paths in the network, which can be guessed on-the-fly in the case of LTL, here we also search for witness flow functions, which have to be guessed in a global manner.
Let \( \{ \varphi_1, \ldots, \varphi_k \} \) be the set of flow state formulas in \( \varphi \). Assume that \( \varphi_1, \ldots, \varphi_k \) are ordered so that for all \( 1 \leq i \leq k \), all the subformulas of \( \varphi_i \) have indices in \( \{1, \ldots, i\} \). Our model-checking procedure labels \( N \) by new atomic propositions \( q_1, \ldots, q_k \) so that for all \( 1 \leq i \leq k \), we have that \( v \models q_i \) iff \( v \models \varphi_i \) (note that since \( \varphi_i \) is closed, satisfaction is independent of a flow function).

Starting with \( i = 1 \), we model check \( \varphi_1 \), label \( N \) with \( q_1 \), and replace the subformula \( \varphi_1 \) in \( \varphi \) by \( q_1 \). Accordingly, when we handle \( \varphi_i \), it is an \( \exists BFL^\star \) or a \( \forall BFL^\star \) formula. That is, it is of the form \( E \xi \) or \( A \xi \), for a Flow-CTL \( \star \) formula \( \xi \). Assume that \( \varphi_i = E \xi \). We guess a flow function \( f : E \rightarrow \mathbb{N} \), and perform CTL \( \star \) model-checking on \( \xi \), evaluating the flow propositions in \( \xi \) according to \( f \). Since guessing \( f \) requires polynomial space, and CTL \( \star \) model checking is in PSPACE, so is handling of \( \varphi_i \) and of all the subformulas.

Hardness in PSPACE follows from the hardness of CTL \( \star \) model checking [37].

Since BFL \( \star \) contains CTL \( \star \), the lower bound in Theorem 10 is immediate. One may wonder whether reasoning about flow networks without atomic propositions, namely when we specify properties of flow only, is simpler. Theorem 11 below shows that this is not the case. Essentially, the proof follows from our ability to encode assignments to atomic propositions by values of flow.

\textbf{Theorem 11.} BFL \( \star \) model checking is PSPACE-complete already for \( \forall LFL_1 \) formulas without atomic propositions.

We also note that when the given formula is in BFL, we cannot avoid the need to guess a flow function, yet once the flow function is guessed, we can verify it in polynomial time. Accordingly, the model-checking problem for BFL is in P\( \text{NP} \). We discuss this point further below.

In practice, a network is typically much bigger than its specification, and its size is the computational bottleneck. In temporal-logic model checking, researchers have analyzed the system complexity of model-checking algorithms, namely the complexity in terms of the system, assuming the specification is of a fixed length. There, the system complexity of LTL and CTL \( \star \) is NLOGSPACE-complete [29, 27]. We prove that, unfortunately, this is not the case of BFL \( \star \). That is, we prove that while the network complexity of the model-checking problem, namely the complexity in terms of the network, does not reach PSPACE, it does require polynomially many calls to an NP oracle. Essentially, each evaluation of a flow quantifier requires such a call. Formally, we have the following.

\textbf{Theorem 12.} The network complexity of BFL \( \star \) is in \( \Delta^P_2 \) (namely, in P\( \text{NP} \)).

\textbf{Proof.} Fixing the length of the formula in the algorithm described in the proof of Theorem 10, we get that \( k \) is fixed, and so is the length of each subformula \( \varphi_i \). Thus, evaluation of \( \varphi_i \) involves a guess of a flow and then model checking of a fixed size Flow-CTL \( \star \) formula, which can be done in time polynomial in the size of the network. Hence, the algorithm from Theorem 10 combined with an NP oracle gives the required network complexity.

While finding the exact network complexity of model checking BFL \( \star \) and its fragments is interesting from a complexity-theoretical point of view, it does not contribute much to our story. Here, we prove NP and co-NP hardness holds already for very restricted fragments. As good news, in Section 7 we point that for the conjunctive fragment, model checking can be performed in polynomial time.\(^2\)

\(^2\) A possible tightening of our analysis is via the complexity class BH, which is based on a Boolean hierarchy over \( \mathbb{NP} \). Essentially, it is the smallest class that contains \( \mathbb{NP} \) and is closed under union,
Theorem 13. The network complexity of $\exists BFL_1^\ast$ and $\forall BFL_1^\ast$ is NP-complete and co-NP-complete, respectively. Hardness applies already to $\exists LFL_1$ and $\forall LFL_1$ without atomic propositions, and to BFL.

Proof. For the upper bound, it is easy to see that one step in the algorithm described in the proof of Theorem 10 (that is, evaluating $\phi_i$ once all its flow state subformulas have been evaluated), when applied to $\phi_i$ of a fixed length is in NP for $\phi_i$ of the form $E\xi$ and in co-NP for $\phi_i$ of the form $A\xi$.

For the lower bound, we prove NP-hardness for $\exists LFL_1$. Co-NP-hardness for $\forall LFL_1$ follows by dualization. We describe a reduction from CNF-SAT. Let $\theta = C_1 \land \ldots \land C_m$ be a CNF formula over the variables $x_1 \ldots x_n$. We assume that every literal in $x_1, \ldots, x_n, \bar{x}_1, \ldots, \bar{x}_n$ appears exactly in $k$ clauses in $\theta$. Indeed, every CNF formula can be converted to such a formula in polynomial time and with a polynomial blowup.

We construct a flow network $N$ and an $\exists LFL_1$ formula $E\psi$ such that $\theta$ is satisfiable iff $N \models E\psi$. The network $N$ is constructed as demonstrated on the right. Let $Z = \{x_1, \ldots, x_n, \bar{x}_1, \ldots, \bar{x}_n\}$. For a literal $z \in Z$ and a clause $C_i$, the network $N$ contains an edge $\langle z, C_i \rangle$ iff the clause $C_i$ contains the literal $z$. Thus, each vertex in $Z$ has exactly $k$ outgoing edges. The capacity of each of these edges is 1. The flow-LTL formula $\psi = kn \land XX(k \lor 0) \land XX \geq 1$.

In the full version, we prove that $\theta$ is satisfiable iff $N \models E\psi$.

Finally, note that $\psi$ does not contain atomic propositions. Also, the same proof holds with the BFL formula $\psi = kn \land AXEXk \land AXAXAX \geq 1$.

5.1 Flow synthesis

In the flow-synthesis problem, we are given a network $N$ and an $\exists BFL_1^\ast$ formula $E\phi$, and we have to return a flow function $f$ with which $\phi$ is satisfied in $N$, or declare that no such function exists. The problem is clearly at least as hard as CTL$^\ast$ model checking. Also, by guessing $f$, its complexity does not go beyond CTL$^\ast$ model-checking complexity. The network complexity of the problem coincides with that of $\exists BFL_1^\ast$ model checking. Thus, we have the following.

Theorem 14. The flow-synthesis problem for $\exists BFL_1^\ast$ is PSPACE-complete, and its network complexity is NP-complete.

6 Model Checking Extensions of BFL$^\ast$

In Section 4, we define several extensions of BFL$^\ast$. In this section we study the model-checking complexity for each of the extensions, and show that they do not require an increase

intersection, and complement. The levels of the hierarchy start with BH$_1 = \text{NP}$, and each level adds internal intersections as well as intersection with a co-NP (even levels) or an NP (odd levels) language [41]. BH is contained in $\Delta^P_2$. It is not hard to prove that the network complexity of the fragment of BFL$^\ast_1$ that contains at most $k$ flow quantifiers is in BH$_{k+1} \cap$ co-BH$_{k+1}$. Indeed, the latter contain problems that are decidable in polynomial time with $k$ parallel queries to an NP oracle [4]. A BH$_k$ lower bound can also be shown.
Flow Logic

in the complexity. The techniques for handling them are, however, richer: For positive path quantification, we have to refine the network and add a path-predicate that specifies positive flow, in a similar way fairness is handled in temporal logics. For maximal-flow quantification, we have to augment the model-checking algorithm by calls to a procedure that finds the maximal flow. For non-integral flow quantification, we have to reduce the model-checking problem to a solution of a linear-programming system. For past operators, we have to extend the model-checking procedure for CTL$^\star$ with branching past. Finally, for first-order quantification over flow values, we have to first bound the range of relevant values, and then apply model checking to all relevant values.

Positive path quantification

Given a network $N$, it is easy to generate a network $N'$ in which we add a vertex in the middle of each edge, and in which the positivity of paths correspond to positive flow in the new intermediate vertices. Formally, assuming that we label the new intermediate vertices by an atomic proposition $edge$, then the BFL$^\star$ path formula $\xi_{positive} = G(edge \rightarrow 0)$ characterizes positive paths, and replacing a state formula $A\psi$ by the formula $A(\xi_{positive} \rightarrow \psi)$ restricts the range of path quantification to positive paths. Now, given a BFL$^\star$ formula $\phi$, it is easy to generate a BFL$^\star$ formula $\phi'$ such that $N \models \phi$ iff $N' \models \phi'$. Indeed, we only have to (recursively) modify path formulas so that vertices labeled $edge$ are ignored: $XX\xi$ is replaced by $XX\xi$, and $\xi_1U\xi_2$ is replaced by $(\xi_1 \vee edge)U(\xi_2 \land \neg edge)$. Hence, the complexity of model-checking is similar to BFL$^\star$.

Maximal flow quantification

The maximal flow $\gamma_{max}$ in a flow network can be found in polynomial time. Our model-checking algorithm for BFL$^\star$ handles each flow state subformula $E\phi$ by guessing a flow function $f : E \rightarrow \mathbb{N}$ with which the Flow-CTL$^\star$ formula $\phi$ holds. For an $E_{max}$ quantifier, we can guess only flow functions for which the flow leaving the source vertex is $\gamma_{max}$. In addition, after calculating the maximal flow, we can substitute $\gamma_{max}$, in formulas that refer to it, by its value. Hence, the complexity of model-checking is similar to that of BFL$^\star$.

Non-integral flow quantification

Recall that our BFL$^\star$ model-checking algorithm handles each flow state subformula $E\phi$ by guessing a flow function $f : E \rightarrow \mathbb{N}$ with which the Flow-CTL$^\star$ formula $\phi$ holds. Moving to non-integral flow functions, the guessed function $f$ should be $f : E \rightarrow \mathbb{R}$, where we cannot bound the size or range of guesses.

Accordingly, in the non-integral case, we guess, for every vertex $v \in V$, an assignment to the flow propositions that appear in $\phi$. Then, we perform two checks. First, that $\phi$ is satisfied with the guessed assignment – this is done by CTL$^\star$ model checking, as in the case of integral flows. Second, that there is a non-integral flow function that satisfies the flow constraints that appear in the vertices. This can be done in polynomial time by solving a system of inequalities [35] (see Lemma 16 for the details in the case of vertex-constrained integral flow functions). Thus, as in the integral case, handling each flow state formula $E\phi$ can be done in PSPACE, and so is the complexity of the entire algorithm.
Past operators

Recall that our algorithm reduces BFL* model checking to a sequence of calls to a CTL* model-checking procedure. Starting with a BFL* formula with past operators, the required calls are to a model-checking procedure for CTL* with past. By [25], model checking CTL* with branching past is PSPACE-complete, and thus so is the complexity of our algorithm.

First-Order quantification on flow values

For a flow network \( N = (AP, V, E, c, \rho, s, T) \), let \( C_N = 1 + \sum_{e \in EC} (\rho(e)) \). Thus, for every flow function \( f \) for \( N \) and for every vertex \( v \in V \), we have \( f(v) < C_N \). We claim that when we reason about BFL* formulas with quantified flow values, we can restrict attention to values in \( \{0, 1, \ldots, C_N\} \).

**Lemma 15.** Let \( N \) be a flow network and let \( \theta = \forall x_1 \varphi \) be a BFL*({+, *}) formula over the variables \( X = \{x_1, \ldots, x_n\} \), and without free variables. Then, \( N \models \theta \) iff \( N \models \varphi[x_1 \leftarrow \gamma] \), for every \( 0 \leq \gamma \leq C_N \).

By Lemma 15, the model-checking problem for BFL*({+, *}) is PSPACE-complete.

7 A Polynomial Fragment

In this section we show that the model-checking problem for CBFL* (see Section 4.6) can be solved in polynomial time.

Our model-checking algorithm reduces the evaluation of a CBFL* formula into a sequence of solutions to the vertex-constrained flow problem. In this problem, we are given a flow network \( N = (AP, V, E, c, \rho, s, T) \) in which each vertex \( v \in V \) is attributed by a range \( [\gamma_l, \gamma_u] \in \mathbb{N} \times (\mathbb{N} \cup \{\infty\}) \). The problem is to decide whether there is a flow function \( f : E \to \mathbb{R} \) such that for all vertices \( v \in V \), we have \( \gamma_l \leq f(v) \leq \gamma_u \).

**Lemma 16.** The vertex-constrained flow problem can be solved in polynomial time. If there is a solution that is a non-integral flow function, then there is also a solution that is an integral flow function, and the algorithm returns such a solution.

**Theorem 17.** CBFL* model checking can be solved in polynomial time.

**Proof.** Let \( N = (AP, V, E, c, \rho, s, T) \), and consider a CBFL* formula \( \varphi \). If \( \varphi \) is in CBFL* \( \leq_1 \), we can clearly label in linear time all the vertices in \( N \) by a fresh atomic proposition \( p_\varphi \) that maintains the satisfaction of \( \varphi \). That is, in all vertices \( v \in V \), we have that \( p_\varphi \in \rho(v) \) iff \( v \models \varphi \). Otherwise, \( \varphi \) is a CBFL* \( \leq_{i+1} \) formula for some \( i \geq 0 \). We show how, assuming that the vertices of \( N \) are labeled by atomic propositions that maintain satisfaction of the subformulas of \( \varphi \) that are CBFL* \( \leq_i \) formulas, we can label them, in polynomial time, by a fresh atomic proposition that maintains the satisfaction of \( \varphi \).

Recall that \( \varphi \) is a Boolean assertion over CBFL* \( \leq_i \) formulas and flow formulas of the form \( \varepsilon(A\psi_1 \land \cdots \land A\psi_n) \), where each \( \psi_j \) is of the form \( X_{k_j} \xi_j \) or \( X_{k_j} G \xi_j \), where \( k_j \geq 0 \) and \( \xi_j \) is a CBFL* \( \leq_i \) formula or a flow proposition (that is, \( >, \gamma, \leq, \gamma, \geq, \), or \( \leq \), for an integer \( \gamma \in \mathbb{N} \)).

Since CBFL* subformulas have already been evaluated, we describe how to evaluate subformulas of the form \( \theta = \varepsilon(A\psi_1 \land \cdots \land A\psi_n) \). Intuitively, since the formulas in \( \theta \) include no disjunctions, they impose constraints on the vertices of \( N \) in a deterministic manner. These constraints can be checked in polynomial time by solving a vertex-constrained flow problem. Recall that for each \( 1 \leq j \leq n \), the formula \( \psi_j \) is of the form \( X_{k_j} \xi_j \) or \( X_{k_j} G \xi_j \), for some \( k_j \geq 0 \), and a CBFL* \( \leq_i \) formula or a flow proposition \( \xi_j \). In order to evaluate \( \theta \) in
a vertex \( v \in V \), we proceed as follows. For each \( 1 \leq j \leq n \), the formula \( \xi_j \) imposes either a Boolean constraint (in case \( \xi_j \) is a \( \text{CBFL}^* \) formula) or a flow constraint (in case \( \xi_j \) is a flow proposition) on a finite subset \( V^v_j \) of \( V \). Indeed, if \( \psi_j = X^{k_j} \xi_j \), then \( V^v_j \) includes all the vertices reachable from \( v \) by a path of length \( k_j \), and if \( \psi_j = X^{k_j} G \xi_j \), then \( V^v_j \) includes all the vertices reachable from \( v \) by a path of length at least \( k_j \). We attribute each vertex by the constraints imposes on it by all the conjuncts in \( \theta \). If one of the Boolean constrains does not hold, then \( \theta \) does not hold in \( v \). Otherwise, we obtain a set of flow constraints for each vertex in \( V \). For example, if \( \theta = E(AXXp \land AX > 5 \land AG \leq 8) \), then in order to check whether \( \theta \) holds in \( s \), we assign the flow constraint \( \leq 8 \) to all the vertices reachable from \( s \), and assign the flow constraint \( > 5 \) to all the successors of the successors of \( s \). If one of these successors of successors does not satisfy \( p \), we can skip the check for a flow and conclude that \( s \) does not satisfy \( \theta \). Otherwise, we search for such a flow, as described below.

The flow constrains for a vertex induce a closed, open, or half-closed range. The upper bound in the range may be infinity. For example, the constrains \( > 6 \), \( < 10 \), \( \leq 8 \) induce the half-closed range \((6, 8]\). Note that it may be that the induced range is empty. For example, the constraints \( \leq 6 \) and \( > 8 \) induce an empty range. Then, \( \theta \) does not hold in \( v \). Since we are interested in integral flows, we can convert all strict bounds to non-strict ones. For example, the range \((6, 8]\) can be converted to \([7, 8]\). Note that since we are interested in integral flow, a non-empty open range may not be satisfiable, and we refer to it as an empty range. For example, the range \((6, 7)\) is empty. Hence, the satisfaction of \( \theta \) in \( v \) is reduced to an instance of the vertex-constrained flow problem. By Lemma 16, deciding whether there is a flow function that satisfies the constraints can be solved in polynomial time.

Remark. Note that the same algorithm can be applied when we consider non-integral flow functions, namely in \( \text{CBFL}^* \) with the \( \text{A}\text{IR} \) flow quantifier. There, the induced vertex-constrained flow problem may include open boundaries. The solution need not be integral, but can be found in polynomial time by solving a system of inequalities [35].

References


Rule Formats for Nominal Process Calculi

Luca Aceto¹, Ignacio Fábregas², Álvaro García-Pérez³, Anna Ingólfsdóttir⁴, and Yolanda Ortega-Mallén⁵

Abstract
The nominal transition systems (NTSs) of Parrow et al. describe the operational semantics of nominal process calculi. We study NTSs in terms of the nominal residual transition systems (NRTRs) that we introduce. We provide rule formats for the specifications of NRTRs that ensure that the associated NRTR is an NTS and apply them to the operational specification of the early pi-calculus. Our study stems from the recent Nominal SOS of Cimini et al. and from earlier works in nominal sets and nominal logic by Gabbay, Pitts and their collaborators.

1998 ACM Subject Classification D.3.1 Formal Definitions and Theory, F.1.1 Models of Computation, F.1.2 Modes of Computation, F.3.1 Specifying and Verifying and Reasoning about Programs, F.3.2 Semantics of Programming Languages

Keywords and phrases nominal sets, nominal structural operational semantics, process algebra, nominal transition systems, scope opening, rule formats

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2017.10

1 Introduction

The goal of this paper is to develop the foundations of a framework for studying the meta-theory of structural operational semantics (SOS) [27] for process calculi with names and name-binding operations, such as the π-calculi [29]. To this end, we build on the large body...
of work on rule formats for SOS, as surveyed in [2, 22], and on the nominal techniques of Gabbay, Pitts and their co-workers [8, 15, 25, 30].

Rule formats provide syntactic templates guaranteeing that the models of the calculi, whose semantics they specify, enjoy some desirable properties. A first design decision that has to be taken in developing a theory of rule formats for a class of languages is therefore the choice of the semantic objects specified by the rules. The target semantic model we adopt in our study is that of nominal transition systems (NTSs), which have been introduced by Parrow et al. in [23, 24] as a uniform model to describe the operational semantics of a variety of calculi with names and name-binding operations. Based on this choice, a basic sanity criterion for a collection of rules describing the operational semantics of a nominal calculus is that they specify an NTS, and we present a rule format guaranteeing this property (Thm. 28).

As a first stepping stone in our study, we introduce nominal residual transition systems (NRTSs), and study NTSs in terms of NRTSs (Section 2). More specifically, NRTSs enjoy one desirable property in the setting of nominal calculi, namely that their transition relation is equivariant (which means that it treats names uniformly). NTSs are NRTSs that, in addition to having an equivariant transition relation, satisfy a property Parrow et al. call alpha-conversion of residuals (see Def. 3 for the details). The latter property formalises a key aspect of calculi in which names can be scoped to represent local resources. To wit, one crucial feature of the π-calculus is scope opening [21]. Consider a transition $p \xrightarrow{\pi(b)} p'$ in which a process $p$ exports a private/local channel name $b$ along channel $a$. Since the name $b$ is local, it ‘can be subject to alpha-conversion’ [23] and the transitions $p \xrightarrow{\pi(c)} p\{b/c\}$ should also be present for each ‘fresh name’ $c$.

In contrast to related work [7, 9], our approach uses nominal terms to connect the specification system with the semantic model. This has the advantage of capturing the requirement that transitions be ‘up to alpha-equivalence’ (typical in nominal calculi) without instrumenting alpha-conversion explicitly in the specification system.

We specify an NRTS by means of a nominal residual transition system specification (NRTSS), which describes the syntax of a nominal calculus in terms of a nominal signature (Section 3) and its semantics by means of a set of inference rules (Section 4). We develop the basic theory of the NRTS/NRTSS framework, building on the nominal algebraic datatypes of Pitts [25] and the nominal rewriting framework of Fernandez and Gabbay [9]. Based on this framework, we provide rule formats [2, 22] for NRTSSs (Section 5) that ensure that the induced transition relation is equivariant (Theorem 22) and enjoys alpha-conversion of residuals (Theorem 28), and is therefore an NTS. Section 6 presents an example of application of our rule formats to the setting of the π-calculus, and Section 7 discusses avenues for future work, as well as related work, and concludes.

## 2 Preliminaries

### Nominal sets

We follow earlier foundational work by Gabbay and Pitts on nominal sets in [17, 25, 26]. We assume a countably infinite set $\mathbb{A}$ of atoms and consider $\text{Perm } \mathbb{A}$ as the group of finite permutations of atoms (hereafter permutations) ranged over by $\pi$, where we write $i$ for the identity, $\circ$ for composition and $\pi^{-1}$ for the inverse of permutation $\pi$. We are particularly interested in transpositions of two atoms: $(a \ b)$ stands for the permutation that swaps $a$ with $b$ and leaves all other atoms fixed. Every permutation $\pi$ is equal to the composition of a finite number of transpositions, i.e. $\pi = (a_1 \ b_1) \circ \ldots \circ (a_n \ b_n)$ with $n \geq 0$. 
An action of the group \( \text{Perm} A \) on a set \( S \) is a binary operation mapping each \( \pi \in \text{Perm} A \) and \( s \in S \) to an element \( \pi \cdot s \in S \), and satisfying the identity law \( \cdot : s = s \) and the composition law \( (\pi_1 \circ \pi_2) \cdot s = \pi_1 \cdot (\pi_2 \cdot s) \). A \( \text{Perm} A \)-set is a set equipped with an action of \( \text{Perm} A \).

We say that a set of atoms \( A \) supports an object \( s \) if \( \pi \cdot s = s \) for every permutation \( \pi \) that leaves each element \( a \in A \) invariant. In particular, we are interested in sets all of whose elements have finite support (Def. 2.2 of [25]).

\[ ▶ \text{Definition 1 (Nominal sets).} \] A nominal set is a \( \text{Perm} A \)-set all of whose elements are finitely supported.

For each element \( s \) of a nominal set, we write \( \text{supp}(s) \) for the least set that supports \( s \). (Intuitively, the action of permutations on a set \( S \) determines that a finitely supported \( s \in S \) only depends on atoms in \( \text{supp}(s) \), and no others.) The set \( H \) of atoms is a nominal set when \( \pi \cdot a = \pi a \) since \( \text{supp}(a) = \{a\} \) for each atom \( a \in H \). The set \( \text{Perm} A \) of finite permutations is also a nominal set where the permutation action on permutations is given by conjugation, i.e. \( \pi \cdot \pi' = \pi \circ \pi' \circ \pi^{-1} \), and the support of a permutation \( \pi \) is \( \text{supp}(\pi) = \{a \mid \pi a \neq a\} \).

Given two \( \text{Perm} A \)-sets \( S \) and \( T \) and a function \( f : S \to T \), the action of permutation \( \pi \) on function \( f \) is given by conjugation, i.e. \( (\pi \cdot f)(s) = \pi \cdot f(\pi^{-1} \cdot s) \) for each \( s \in S \). We say that a function \( f : S \to T \) is equivariant iff \( \pi \cdot f(s) = f(\pi \cdot s) \) for every \( \pi \in \text{Perm} A \) and every \( s \in S \). The intuition is that an equivariant function \( f \) is atom-blind, in that \( f \) does not treat any atom preferentially. It turns out that a function \( f \) is equivariant iff \( \text{supp}(f) = \emptyset \) (Rem. 2.13 of [25]). The function \( \text{supp} \) is equivariant (Prop. 2.11 of [25]).

Let \( S \) be a \( \text{Perm} A \)-set, we write \( S_b \) for the nominal set that contains the elements in \( S \) that are finitely supported. The nominal function set between nominal sets \( S \) and \( T \) is the nominal set \( S \times T \) of finitely supported functions from \( S \) to \( T \)—be they equivariant or not. Let \( S_1 \) and \( S_2 \) be nominal sets. The product \( S_1 \times S_2 \) is a nominal set (Prop. 2.14 of [25]). The permutation action for products is given componentwise (Eq (1.12) of [25]).

An element \( s_1 \in S_1 \) is fresh in \( s_2 \in S_2 \), written \( s_1 \neq s_2 \), iff \( \text{supp}(s_1) \cap \text{supp}(s_2) = \emptyset \). The freshness relation is equivariant (Eq. (3.2) of [25]).

Finally, we consider atom abstractions (Section 4 of [25]), which represent alpha-equivalence classes of elements.

\[ ▶ \text{Definition 2 (Atom abstraction).} \] Given a nominal set \( S \), the atom abstraction of atom \( a \) in element \( s \in S \), written \( \langle a \rangle s \), is the \( \text{Perm} A \)-set \( \langle a \rangle s = \{ (b, (ba) \cdot s) \mid b = a \lor b \neq s \} \), whose permutation action is \( \pi \cdot (a \cdot s) = \{ (\pi \cdot b, \pi \cdot ((ba) \cdot s)) \mid \pi \cdot b = \pi \cdot a \lor \pi \cdot b \neq \pi \cdot s \} \).

We write \( [A] S \) for the set of atom abstractions in elements of \( S \), which is a nominal set (Def. 4.4 of [25]), since \( \text{supp}(\langle a \rangle s) = \text{supp}(s) \setminus \{a\} \) for each atom \( a \) and element \( s \in S \).

**Nominal Transition Systems**

Nominal transitions systems adopt the state/residual presentation for transitions of [4], where a residual is a pair consisting of an action and a state. In [23], Parrow et al. develop modal logics for process algebras à la Hennessy-Milner. Here we are mainly interested in the transition relation and we adapt Definition 1 in [23] by removing the predicates. We write \( \mathcal{P}_\pi(A) \) for the finite power set of \( A \).

\[ ▶ \text{Definition 3 (Nominal transition system).} \] A nominal transition system (NTS) is a quadruple \( (S, \text{Act}, bn, \rightarrow) \) where \( S \) and \( \text{Act} \) are nominal sets of states and actions respectively, \( bn : \text{Act} \to \mathcal{P}_\pi(A) \) is an equivariant function that delivers the binding names in an action, and \( \rightarrow \subseteq S \times (\text{Act} \times S) \) is an equivariant binary transition relation from states to residuals.
(we let $Act \times S$ be the set of residuals). The function $bn$ is such that $bn(\ell) \subseteq \text{supp}(\ell)$ for each $\ell \in Act$. We often write $p \rightarrow (\ell, p')$ in lieu of $(p, (\ell, p')) \in \rightarrow$.

Finally, the transition relation $\rightarrow$ must satisfy alpha-conversion of residuals, that is, if $a \in bn(\ell)$, $b \neq (\ell, p')$ and $p \rightarrow (\ell, p')$ then also $p \rightarrow ((a \cdot b) \cdot (\ell, p'))$, or equivalently $p \rightarrow (a \cdot b) \cdot (\ell, p')$.

We will consider an NTS (without its associated binding-names function $bn$) as a particular case of a nominal residual transition system, which we introduce next.

**Definition 4** (Nominal residual transition system). A nominal residual transition system (NRTS) is a triple $(S, R, \rightarrow)$ where $S$ and $R$ are nominal sets, and where $\rightarrow \subseteq S \times R$ is an equiavariant binary transition relation. We say $S$ is the set of states and $R$ is the set of residuals.

The connection between NTSs and NRTSs will be explained in more detail in Section 5.

## 3 Nominal terms

This section is devoted to the notion of nominal terms, which are syntactic objects that make use of the atom abstractions of Definition 2 and represent terms up to alpha-equivalence. As a first step, we introduce raw terms, devoid of any notion of alpha-equivalence. Our raw terms resemble those from the literature, mainly [8, 9, 25, 30], but with some important differences. In particular, our terms include both variables (i.e. unknowns) and moderated terms (i.e. explicit permutations over raw terms), and we consider atom and abstraction sorts. (The raw terms of [25] do not include moderated terms, and the ones in [9, 30] only consider moderated variables. In [8] the authors consider neither atom nor abstraction sorts.) We also adopt the classic presentation of free algebras and term algebras in [6, 18] in a different way from that in [8, 25]. The raw terms correspond to the standard notion of free algebra over a signature generated by a set of variables. We then adapt the $\Sigma$-structures of [8] to our sorting schema. Finally, the nominal terms are the interpretations of the ground terms in the initial $\Sigma$-structure; they coincide with the nominal algebraic terms of [25].

**Definition 5** (Nominal signature and nominal sort). A nominal signature (or simply a signature) $\Sigma$ is a triple $(\Delta, A, F)$ where $\Delta = \{\delta_1, \ldots, \delta_n\}$ is a finite set of base sorts, $A$ is a countable set of atom sorts, and $F$ is a finite set of function symbols. The nominal sorts over $\Delta$ and $A$ are given by the grammar $\sigma :: \delta | \alpha | [\alpha] \sigma | \sigma_1 \times \ldots \times \sigma_k$, with $k \geq 0$, $\delta \in \Delta$ and $\alpha \in A$. The sort $[\alpha] \sigma$ is the abstraction sort. Symbol $\times$ denotes the product sort, which is associative; $\sigma_1 \times \ldots \times \sigma_k$ stands for the sort of the empty product when $k = 0$, which we may write as $1$. We write $S$ for the set of nominal sorts. We arrange the function symbols in $F$ based on the sort of the data that they produce. We write $f_{ij} \in F$ with $1 \leq i \leq n$ and $1 \leq j \leq m_i$ such that $f_{ij}$ has arity $\sigma_{ij} \rightarrow \delta_i$, where $\delta_i$ is a base sort.

The theory of nominal sets extends to the case of (countably) many-sorted atoms (see Section 4.7 in [25]). We assume that $A$ contains a countably infinite collection of atoms $a_{\alpha}, b_{\alpha}, c_{\alpha}, \ldots$ for each atom sort $\alpha$ such that the sets of atoms $A_\alpha$ of each sort are mutually disjoint. We write $\text{Perm}_s A = \{\pi \in \text{Perm} A \mid \forall \alpha \in A. \forall a \in A_\alpha. \pi a \in A_\alpha\}$ for the subgroup of finite permutations that respect the sorting. The sorted nominal sets are the $\text{Perm}_s A$-sets whose elements are finitely supported. In the sequel we may drop the $s$ subscript in $\text{Perm}_s A$ and omit the ‘sorted’ epithet from ‘sorted nominal sets’.

We let $V$ be a set that contains a countably infinite collection of variable names (variables for short) $x_{\sigma}, y_{\sigma}, z_{\sigma}, \ldots$ for each sort $\sigma$, such that the sets of variables $V_{\sigma}$ of each sort are mutually disjoint. We also assume that $V$ is disjoint from $A$. 


Definition 6 (Raw terms). Let \( \Sigma = (\Delta, A, F) \) be a signature. The set of raw terms over signature \( \Sigma \) and set of variables \( \mathcal{V} \) (raw terms for short) is given by the grammar

\[
\tau_\sigma ::= x_\sigma \mid a_\sigma \mid (\pi \cdot \tau_\sigma)_{\sigma} \mid ([a_\sigma]t_\sigma)_{|a_\sigma} \mid (t_{\sigma_1}, \ldots, t_{\sigma_k})_{\sigma_1 \times \ldots \times \sigma_k} \mid (f_{ij}(t_{\sigma_j}))_{s_i},
\]

where term \( x_\sigma \) is a variable of sort \( \sigma \), term \( a_\sigma \) is an atom of sort \( \alpha \), term \( (\pi \cdot \tau_\sigma)_{\sigma} \) is a moderated term (i.e., the explicit, or delayed, permutation \( \pi \) over term \( \tau_\sigma \), term \( ([a_\sigma]t_\sigma)_{|a_\sigma} \) is the abstraction of atom \( a_\sigma \) in term \( \tau_\sigma \), term \( (t_{\sigma_1}, \ldots, t_{\sigma_k})_{\sigma_1 \times \ldots \times \sigma_k} \) is the product of terms \( t_{\sigma_1}, \ldots, t_{\sigma_k} \), and term \( (f_{ij}(t_{\sigma_j}))_{s_i} \) is the datum of base sort \( \delta_i \) constructed from term \( t_{\sigma_j} \) and function symbol \( f_{ij} : \delta_i \rightarrow \delta_j \). When they are clear from the context or immaterial, we leave the arities and sorts implicit and write \( x, a, \pi \cdot t, [a]t, (t_1, \ldots, t_k), f(t), \) etc.

The raw terms are the inhabitants of the carrier of the free algebra over the set of variables \( \mathcal{V} \) and over the \( \mathcal{S} \)-sorted conventional signature that consists of the function symbols in \( F \), together with a constant symbol for each atom \( a_\alpha \), a unary symbol that produces moderated terms for each permutation \( \pi \) and each sort \( \sigma \), a unary symbol that produces abstractions for each atom \( a_\alpha \) and sort \( \sigma \), and a \( k \)-ary symbol that produces a product of sort \( \sigma_1 \times \ldots \times \sigma_k \) for each sequence of sorts \( \sigma_1, \ldots, \sigma_k \). (See [18] for a classic presentation of term algebras, initial algebra semantics and free algebras.)

We write \( T(\Sigma, \mathcal{V})_\sigma \) for the set of raw terms of sort \( \sigma \). A raw term \( t \) is ground iff no variables occur in \( t \). We write \( T(\Sigma)_\sigma \) for the set of ground terms of sort \( \sigma \). The sets of raw terms (resp., ground terms) of each sort are mutually disjoint as terms carry sort information. Therefore we sometimes identify the family \( \{ T(\Sigma, \mathcal{V})_\sigma \}_{\sigma \in \mathcal{S}} \) of \( \mathcal{S} \)-indexed raw terms and the family \( \{ T(\Sigma)_\sigma \}_{\sigma \in \mathcal{S}} \) of \( \mathcal{S} \)-indexed ground terms with their respective ranges \( \bigcup_{\sigma \in \mathcal{S}} T(\Sigma, \mathcal{V})_\sigma \) and \( \bigcup_{\sigma \in \mathcal{S}} T(\Sigma)_\sigma \), which we abbreviate as \( T(\Sigma, \mathcal{V}) \) and \( T(\Sigma) \) respectively.

The set \( T(\Sigma, \mathcal{V}) \) of raw terms is a nominal set, with the Term \( \mathcal{A} \)-action and the support of a raw term given by:

\[
\begin{align*}
\pi \cdot x &= x & \text{supp}(x) &= \emptyset \\
\pi \cdot a &= \pi a & \text{supp}(a) &= \{a\} \\
\pi \cdot (\pi_1 \cdot t) &= (\pi \cdot \pi_1) \cdot (\pi \cdot t) & \text{supp}(\pi \cdot t) &= \text{supp}(\pi) \cup \text{supp}(t) \\
\pi \cdot [a]t &= [\pi a](\pi \cdot t) & \text{supp}([a]t) &= \{a\} \cup \text{supp}(t) \\
\pi \cdot (t_1, \ldots, t_k) &= (\pi \cdot t_1, \ldots, \pi \cdot t_k) & \text{supp}(t_1, \ldots, t_k) &= \text{supp}(t_1) \cup \ldots \cup \text{supp}(t_k) \\
\pi \cdot (f(t)) &= f(\pi \cdot t), & \text{supp}(f(t)) &= \text{supp}(t).
\end{align*}
\]

It is straightforward to check that the permutation action for raw terms is sort-preserving (remember that permutations are also sort-preserving). The set \( T(\Sigma) \) of ground terms is also a nominal set since it is closed with respect to the Term \( \mathcal{A} \)-action given above.

Example 7 (\( \pi \)-calculus). Consider a signature \( \Sigma \) for the \( \pi \)-calculus [7, 29] given by a single atom sort \( \text{ch} \) of channel names, and base sorts \( \text{pr} \) and \( \text{ac} \) for processes and actions respectively. The function symbols (adapted from [29]) are the following:

\[
F = \{ \text{null} : 1 \rightarrow \text{pr}, \quad \text{par} : (\text{pr} \times \text{pr}) \rightarrow \text{pr}, \quad \text{tau} : \text{pr} \rightarrow \text{pr}, \quad \text{sum} : (\text{pr} \times \text{pr}) \rightarrow \text{pr}, \quad \text{in} : (\text{ch} \times \text{ch}) \rightarrow \text{pr}, \quad \text{rep} : \text{pr} \rightarrow \text{pr}, \quad \text{out} : (\text{ch} \times \text{ch}) \rightarrow \text{ac}, \quad \text{new} : [\text{ch}] \rightarrow \text{pr}, \quad \text{bout} : (\text{ch} \times \text{ch}) \rightarrow \text{ac} \}.
\]

Recalling terminology from [7, 29], \( \text{null} \) stands for inaction, \( \text{tau}(p) \) for the internal action after which process \( p \) follows, \( \text{in}(a, [b]p) \) for the input at channel \( a \) where the input name is bound to \( b \) in the process \( p \) that follows, \( \text{out}(a, b, p) \) for the output of name \( b \) through
channel $a$ after which process $p$ follows, $\text{par}(p,q)$ for parallel composition, $\text{sum}(p,q)$ for nondeterministic choice, $\text{rep}(p)$ for parallel replication, and $\text{new}(a[p]$ for the restriction of channel $a$ in process $p$ (a is private in $p$). Actions and processes belong to different sorts. We use $\tau A$, $\text{out}(a,b)$, $\text{in}(a,b)$ and $\text{bout}(a,b)$ respectively for the internal action, the output action, the input action and the bound output action.

The set of terms of the $\pi$-calculus corresponds to the subset of ground terms over $\Sigma$ of sort $\text{pr}$ and $\text{ac}$ in which no moderated (sub-)terms occur. For instance, the process $(\nu b)(\pi b.0)$ corresponds to the ground term $\text{new}([b]([\text{out}(a,b,\text{null})]))$, whose support is $\{a,b\}$. Both free and bound channel names (such as the $a$ and $b$ respectively in the example process) are represented by atoms. The set of ground terms also contains generalised processes and actions with moderated (sub-)terms $\pi \bullet p$, which stand for a delayed permutation $\pi$ that ought to be applied to a term $p$, e.g. $\text{new}(\pi \bullet ([b]([\text{out}(a,b,\text{null})])))$.

Raw terms allow variables to occur in the place of any ground subterm. The variables represent unknowns, and should be mistaken neither with free nor bound channel names. For instance, the raw term $\text{new}([b]([\text{out}(a,b,x)])$ represents a process $(\nu b)(\pi b. P)$ where the $x$ is akin to the meta-variable $P$, which stands for some unknown process. The process $(\nu b)(\pi b. P)$ unifies with $(\nu b)(\pi b.0)$ by replacing $P$ with 0. In the nominal setting, the raw term $\text{new}([b]([\text{out}(a,b,x)]))$ unifies with ground term $\text{new}([b]([\text{out}(a,b,\text{null})]))$, by means of a substitution $\varphi$ such that $\varphi(x) = \text{null}$. Formally, substitutions are defined below.

**Definition 8 (Substitution).** A substitution $\varphi : \mathcal{V} \rightarrow \text{b}\mathcal{T}(\Sigma, \mathcal{V})$ is a sort-preserving, finitely supported function from variables to raw terms. The domain $\text{dom}(\varphi)$ of a substitution $\varphi$ is the set $\{ x \mid \varphi(x) \neq x \}$. A substitution $\varphi$ is ground iff $\varphi(x) \in \mathcal{T}(\Sigma)$ for every variable $x \in \text{dom}(\varphi)$.

The set of substitutions is a nominal set. The extension to raw terms $\overline{\varphi}$ of substitution $\varphi$ is the unique homomorphism induced by $\varphi$ from the free algebra $\mathcal{T}(\Sigma, \mathcal{V})$ to itself, which coincides with the function given by:

\[
\overline{\varphi}(x) = \varphi(x) \quad \overline{\varphi}([a]t) = [a](\overline{\varphi}(t)) \\
\overline{\varphi}(a) = a \quad \overline{\varphi}(t_1, \ldots, t_k) = (\overline{\varphi}(t_1), \ldots, \overline{\varphi}(t_k)) \\
\overline{\varphi}(\pi \bullet t) = \pi \bullet \overline{\varphi}(t) \quad \overline{\varphi}(f(t)) = f(\overline{\varphi}(t)).
\]

Given substitutions $\varphi$ and $\gamma$ we write $\varphi \circ \gamma$ for their composition, which is defined as follows: For every variable $x$, $(\varphi \circ \gamma)(x) = \overline{\varphi}(\gamma(x))$ where $\gamma(x) = t$. It is straightforward to check that $(\overline{\varphi \circ \gamma})(t) = \overline{\varphi}(\gamma(t))$. We note that our definition of substitution is different form those in both [8, 30], where the authors consider a function that performs the delayed permutations of the moderated terms on-the-fly.

**Lemma 9 (Extension to raw terms is equivariant).** Let $\varphi$ be a substitution and $\pi$ a permutation. Then, $\pi \circ \overline{\varphi} = \overline{\pi \circ \varphi}$.

It is straightforward to check that the support of $\overline{\varphi}$ coincides with the support of $\varphi$. By the above lemma, the set of extended substitutions is also a nominal set, since it is closed with respect to the Perm $A$-action. Hereafter we sometimes write $\varphi(t)$, where $t$ is a raw term, instead of $\overline{\varphi}(t)$. We may also write $\varphi^\pi$ instead of $\pi \circ \varphi$ or $\overline{\pi \circ \varphi}$ for short.

The following result highlights the relation between substitution and the permutation action.

**Lemma 10 (Substitution and permutation action).** Let $\varphi$ be a substitution, $\pi$ a permutation and $t$ a raw term. Then, $\pi \circ \varphi(t) = \varphi^\pi(\pi \circ t)$. 

Our goal is to give meaning to ground terms in nominal sets. To this end, we need a suitable class of algebraic structures that can be used to give an interpretation of those ground terms.

**Definition 11 (Σ-structure).** Let $\Sigma = (\Delta, A, F)$ be a signature. A $\Sigma$-structure $M$ consists of a nominal set $M[\sigma]$ for each sort $\sigma$ defined as follows

\[
\begin{align*}
M[Î»] &= a_

M[Î»\sigma] &= [A_\alpha](M[\sigma]) \\
M[\sigma_1 \times \ldots \times \sigma_k] &= M[\sigma_1] \times \ldots \times M[\sigma_k],
\end{align*}
\]

where the $M[\delta_i]$ with $\delta_i \in \Delta$ are given, as well as an equivariant function $M[f_{ij}] : M[\sigma_{ij}] \rightarrow M[\delta_i]$ for each symbol $(f_{ij})_{\sigma_{ij}\rightarrow\delta_i} \in F$.

The notion of $\Sigma$-structure adapts that of $\Sigma$-structure in [8] to our sorting convention with atom and abstraction sorts. The $\Sigma$-structures characterise a range of interpretations of ground terms into elements of nominal sets, such that any sort $\sigma$ gives rise to the expected nominal set, i.e. atom sorts give rise to sets of atoms, abstraction sorts give rise to sets of atom abstractions, and product sorts give rise to finite products of nominal sets.

Next we define the interpretation of a ground term in a $\Sigma$-structure, which resembles the value of a term in [8].

**Definition 12 (Interpretation of ground terms in a $\Sigma$-structure).** Let $\Sigma$ be a signature and $M$ be a $\Sigma$-structure. The interpretation $M[\]p\]$ of a ground term $p$ in $M$ is given by:

\[
\begin{align*}
M[Î»] &= a & M[(p_1, \ldots, p_k)] &= (M[p_1], \ldots, M[p_k]) \\
M[\pi \cdot p] &= \pi \cdot M[p] & M[f(p)] &= M[f](M[p]) \\
M[\pi p] &= (a)(M[p])
\end{align*}
\]

The next lemma states that interpretation in a $\Sigma$-structure is equivariant and highlights the relation between interpretation and moderated terms.

**Lemma 13 (Interpretation and moderated terms).** Let $M$ be a $\Sigma$-structure. Interpretation in $M$ is equivariant, that is, $\pi \cdot M[\]p\] = M[\pi \cdot p]$ for every ground term $p$ and permutation $\pi$. Moreover, $M[\pi \cdot p] = M[\pi p]$.

Finally, we introduce the $\Sigma$-structure $NT$, which formalises the set of nominal terms.

**Definition 14 (Σ-structure for nominal terms).** Let $\Sigma$ be a signature. The $\Sigma$-structure $NT$ for nominal terms is given by the least tuple $(NT[\delta_1], \ldots, NT[\delta_n])$ satisfying

\[
NT[\delta_i] = NT[\sigma_{i1}] + \ldots + NT[\sigma_{in_i}] \quad \text{for each base sort } \delta_i \in \Delta,
\]

\[
NT[f_{ij}] = \text{inj}_j : NT[\sigma_{ij}] \rightarrow NT[\delta_i], \quad \text{for each function symbol } f_{ij} \in F.
\]

In the conditions above, the ‘less than or equal to’ relation for tuples is pointwise set inclusion. The $NT[f_{ij}]$ is the $j$th injection of the $i$th component in $(NT[\delta_1], \ldots, NT[\delta_n])$.

Nominal terms represent alpha-equivalence classes of raw terms by using the atom abstractions of Definition 2.

**Definition 15 (Nominal terms).** Let $\Sigma$ be a signature. The set $N(\Sigma)_\sigma$ of nominal terms over $\Sigma$ of sort $\sigma$ is the domain of interpretation of the ground terms of sort $\sigma$ in the $\Sigma$-structure $NT$, that is, $N(\Sigma)_\sigma = NT[\sigma]$. 
We sometimes write $p$, $\ell$ instead of $NT[p], \ NT[\ell]$ when it is clear from the context that we are referring to the interpretation into nominal terms of ground terms $p$ and $\ell$.

It can be checked that the nominal sets $N(\Sigma)_\sigma$ coincide (up to isomorphism) with the nominal algebraic datatypes of Definition 8.9 in [25], and therefore by Theorem 8.15 in [25] the nominal terms represent alpha-equivalence classes of raw terms.

4 Specifications of NRTSs

The NRTSs of Definition 4 are meant to be a model of computation for calculi with name-binding operators and state/residual presentation. In this section we present syntactic specifications for NRTSs. We start by defining nominal residual signatures.

Definition 16 (Nominal residual signature). A nominal residual signature (a residual signature for short) is a quintuple $\Sigma = (\Delta, A, \sigma, \rho, F)$ such that $(\Delta, A, F)$ is a nominal signature and $\sigma$ and $\rho$ are distinguished nominal sorts over $\Delta$ and $A$, which we call state sort and residual sort respectively. We say that $N(\Sigma)_\sigma$ is the set of states and $N(\Sigma)_\rho$ is the set of residuals.

Let $T = (S, R, \rightarrow)$ be an NRTS and $\Sigma = (\Delta, A, \sigma, \rho, F)$ be a residual signature. We say that $T$ is an NRTS over signature $\Sigma$ iff the sets of states $S$ and residuals $R$ coincide with the sets of nominal terms of state sort $N(\Sigma)_\sigma$ and residual sort $N(\Sigma)_\rho$ respectively.

Our next goal is to introduce syntactic specifications of NRTSs, which we call nominal residual transition system specifications. To this end, we will make use of residual formulas and freshness assertions over raw terms, which are defined below.

Definition 17 (Residual formula and freshness assertion). A residual formula (a formula for short) over a residual signature $\Sigma$ is a pair $(s, r)$, where $s \in T(\Sigma, V)_\sigma$ and $r \in T(\Sigma, V)_\rho$. We use the more suggestive $s \rightarrow r$ in lieu of $(s, r)$. A formula $s \rightarrow r$ is ground iff $s$ and $r$ are ground terms.

A freshness assertion (an assertion for short) over a signature $\Sigma$ is a pair $(a, t)$ where $a \in A$ and $t \in T(\Sigma, V)$. We will write $a \not\approx t$ in lieu of $(a, t)$. An assertion is ground iff $t$ is a ground term.

Remark. Formulas and assertions are raw syntactic objects, similar to raw terms, which will occur in the rules of the nominal residual transition system specifications to be defined, and whose purpose is to represent respectively transitions and freshness relations involving nominal terms. A formula $s \rightarrow r$ (resp. an assertion $a \not\approx t$) unifies with a ground formula $\varphi(s) \rightarrow \varphi(r)$ (resp. a ground assertion $a \not\approx \varphi(t)$), which in turn represents a transition $NT[\varphi(s)] \rightarrow NT[\varphi(r)]$ (resp. a freshness relation $a\#NT[\varphi(t)]$). For the assertions, notice how the symbols $\not\approx$, $\not\#$ and $\not\#NT$ interact. The ground assertion $a \not\# [a]a$ represents the freshness relation $a\#NT[[a]a]$, which is true. On the other hand, the freshness relation $a\#[a]a$ is false because $a \in \text{supp}([a]a)$.

Permutation action and substitution extend to formulas and assertions in the expected way. Formulas and assertions are elements of nominal sets. Their support is the union of the supports of the raw terms in them, hence we write $\text{supp}(t \rightarrow t')$ and $\text{supp}(a \not\approx t)$. We will also write $b\#(t \rightarrow t')$ and $b\#(a \not\approx t)$ for freshness relations involving formulas and assertions respectively.
Definition 18 (Nominal residual transition system specification). Let \( \Sigma \) be a residual signature \((\Delta, A, \sigma, \rho, F)\). A transition rule over \( \Sigma \) (a rule, for short) is of the form

\[
\frac{\{u_i \rightarrow u'_i \mid i \in I\} \quad \{a_j \not\approx v_j \mid j \in J\}}{t \rightarrow t'},
\]

abbreviated as \( H, \nabla / t \rightarrow t' \), where \( H = \{u_i \rightarrow u'_i \mid i \in I\} \) is a finitely supported set of formulas over \( \Sigma \) (we call \( H \) the set of premisses) and where \( \nabla = \{a_j \not\approx v_j \mid j \in J\} \) is a finite set of assertions over \( \Sigma \) (we call \( \nabla \) the freshness environment). We say formula \( t \rightarrow t' \) over \( \Sigma \) is the conclusion, where \( t \) is the source and \( t' \) is the target. A rule is an axiom iff it has an empty set of premisses. Note that axioms might have a non-empty freshness environment.

A nominal residual transition system specification over \( \Sigma \) (abbreviated to NRTSS) is a set of transition rules over \( \Sigma \).

Permutation action and substitution extend to rules in the expected way; they are applied to each of the formulas and freshness assertions in the rule.

Notice that the rules of an NRTSS are elements of a nominal set. The support of a rule \( H, \nabla / t \rightarrow t' \) is the union of the support of \( H \), the support of \( \nabla \) and the support of \( t \rightarrow t' \). In the sequel we write \( \text{supp}(Ru) \) for the support of rule \( Ru \), and \( a \# Ru \) for a freshness relation involving atom \( a \) and rule \( Ru \). Observe that the set \( H \) of premises of a rule may be infinite, but its support must be finite. However, the freshness environment \( \nabla \) must be finite in order to make the simplification rules of Definition 23 to follow terminating. These simplification rules will be used in Section 5 to define the rule format in Definition 27.

Let \( R \) be an NRTSS. We say that the formula \( s \rightarrow r \) unifies with rule \( Ru \) in \( R \) iff \( Ru \) has conclusion \( t \rightarrow t' \) and \( s \rightarrow r \) is a substitution instance of \( t \rightarrow t' \). If \( s \) and \( r \) are ground terms, we also say that transition \( NT[s] \rightarrow NT[r] \) unifies with \( Ru \).

Definition 19 (Proof tree). Let \( \Sigma \) be a residual signature and \( R \) be an NRTSS over \( \Sigma \). A proof tree in \( R \) of a transition \( NT[s] \rightarrow NT[r] \) is an upwardly branching rooted tree without paths of infinite length whose nodes are labelled by transitions such that

(i) the root is labelled by \( NT[s] \rightarrow NT[r] \), and

(ii) if \( K = \{NT[q_i] \rightarrow NT[q'_i] \mid i \in I\} \) is the set of labels of the nodes directly above a node with label \( NT[p] \rightarrow NT[p'] \), then there exist a rule

\[
\frac{\{u_i \rightarrow u'_i \mid i \in I\} \quad \{a_j \not\approx v_j \mid j \in J\}}{t \rightarrow t'}
\]

in \( R \) and a ground substitution \( \varphi \) such that \( \varphi(t \rightarrow t') = p \rightarrow p' \) and, for each \( i \in I \) and for each \( j \in J \), \( \varphi(u_i \rightarrow u'_i) = q_i \rightarrow q'_i \) and \( a_j \not\approx NT[\varphi(v_j)] \) hold.

We say that \( NT[s] \rightarrow NT[r] \) is provable in \( R \) iff it has a proof tree in \( R \). The transition relation specified by \( R \) consists of all the transitions that are provable in \( R \).

The nodes of a proof tree are labelled by transitions, which contain nominal terms (i.e. syntactic objects that use the atom abstractions of Definition 2). The use of nominal terms captures the convention in typical nominal calculi of considering terms ‘up to alpha-equivalence’.

The fact that the nodes of a proof tree are labelled by nominal terms is the main difference between our approach and previous work in nominal structural operational semantics [1], nominal rewriting [9, 30] and nominal algebra [15]. In all these works, the ‘up-to-alpha-equivalence’ transitions are explicitly instrumented within the model of computation by adding to the specification system inference rules that perform alpha-conversion of raw terms.
5 Rule formats for NRTSSs

This section defines two rule formats for NRTSSs that ensure that:
(i) an NRTSS induces an equivariant transition relation, and thus an NRTS of Definition 4:
(ii) an NRTSS induces a transition relation which, together with an equivariant function
\( \pi \), corresponds to an NTS of Definition 3 [23]. For the latter, we need to ensure that
the induced transition relation is equivariant and satisfies \textit{alpha-conversion of residuals}
(recall, if \( p \rightarrow (\ell, p') \) is provable in \( R \) and \( a \) is in the set of binding names of \( \ell \), then for
every atom \( b \) that is fresh in \( (\ell, p') \) the transition \( p \rightarrow (a b) \cdot (\ell, p') \) is also provable).

As a first step, we introduce a rule format ensuring equivariance of the induced transition
relation.

\begin{itemize}
\item \textbf{Definition 20} (Equivariant format). Let \( R \) be an NRTSS. \( R \) is in \textit{equivariant format}
iff the rule \( (a b) \cdot Ru \) is in \( R \), for every rule \( Ru \) in \( R \) and for each \( a, b \in A \).
\item \textbf{Lemma 21}. Let \( R \) be an NRTSS in equivariant format. For every rule \( Ru \) in \( R \) and for
every permutation \( \pi \), the rule \( \pi \cdot Ru \) is in \( R \).
\item \textbf{Theorem 22} (Rule format for NRTSSs). Let \( R \) be an NRTSS. If \( R \) is in equivariant format
then \( R \) induces an NRTS.
\end{itemize}

Before introducing a rule format ensuring alpha-conversion of residuals, we adapt to our
freshness environments the simplification rules and the entailment relation of Definition 10
and Lemma 15 in [9], which we will use in the definition of the rule format.

\begin{itemize}
\item \textbf{Definition 23} (Simplification of freshness environments). Consider a signature \( \Sigma \). The
following rules, where \( a, b \) are assumed to be distinct atoms and \( \nabla \) is a freshness environment
over \( \Sigma \), define \textit{simplification of freshness environments}:
\[ \{a \not\equiv b\} \cup \nabla \implies \nabla \quad \{a \not\equiv (p_1, \ldots, p_k)\} \cup \nabla \implies \{a \not\equiv p_1, \ldots, a \not\equiv p_k\} \cup \nabla \]
\[ \{a \not\equiv \pi \cdot t\} \cup \nabla \implies \{\pi^{-1} \cdot a \not\equiv t\} \cup \nabla \quad \{a \not\equiv [a]p\} \cup \nabla \implies \nabla \]
\[ \{a \not\equiv [b]p\} \cup \nabla \implies \{a \not\equiv p\} \cup \nabla \quad \{a \not\equiv f(p)\} \cup \nabla \implies \{a \not\equiv p\} \cup \nabla . \]
\end{itemize}

The rules define a reduction relation on freshness environments. We write \( \nabla \Longrightarrow \nabla' \) when
\( \nabla' \) is obtained from \( \nabla \) by applying one simplification rule, and \( \Longrightarrow^* \) for the reflexive and
transitive closure of \( \Longrightarrow \).

\begin{itemize}
\item \textbf{Lemma 24}. The relation \( \Longrightarrow \) is confluent and terminating.
\end{itemize}

A freshness assertion is \textit{reduced} iff it is of the form \( a \not\equiv a \) or \( a \not\equiv x \). We say that \( a \not\equiv a \)
is \textit{inconsistent} and \( a \not\equiv x \) is \textit{consistent}. An environment \( \nabla \) is \textit{reduced} iff it consists only
of reduced assertions. An environment containing a freshness assertion that is not reduced
can always be simplified using one of the rules in Definition 23. Therefore, by Lemma 24,
an environment \( \nabla \) reduces by \( \Longrightarrow^* \) to a unique reduced environment, which we call the
\textit{normal form} of \( \nabla \), written \( \langle \nabla \rangle_{nf} \). An environment \( \nabla \) is \textit{inconsistent} iff \( \langle \nabla \rangle_{nf} \) contains some
inconsistent assertion. We say \( \nabla \) \textit{entails} \( \nabla' \) (written \( \nabla \vdash \nabla' \)) iff either \( \nabla \) is an inconsistent
environment, or \( \langle \nabla' \rangle_{nf} \subseteq \langle \nabla \rangle_{nf} \). We write \( \vdash \nabla \) iff \( \emptyset \vdash \nabla \).

\begin{itemize}
\item \textbf{Lemma 25}. Let \( \nabla \) be an environment over \( \Sigma \). Then, for every ground substitution \( \varphi \), the
conjunction of the freshness relations represented by \( \varphi(\langle \nabla \rangle_{nf}) \) holds iff the conjunction of
the freshness relations represented by \( \varphi(\nabla) \) hold.
\end{itemize}
In particular, if \( \Gamma \vdash \varphi \) then for every ground substitution \( \varphi \) the freshness relations represented by \( \varphi(\varphi) \) hold.

We are interested in NTS [23], which consider signatures with base sorts \( \text{ac} \) and \( \text{pr} \), with a single atom sort \( \text{ch} \) and with source and residual sorts \( \text{pr} \) and \( \text{ac} \times \text{pr} \) respectively. We let \( \Sigma_{\text{NTS}} \) be any such signature parametric on a set \( F \) of function symbols that we keep implicit. We let \( \text{bn} : N(\Sigma)_{\text{ac}} \to \mathcal{P}(\text{ch}) \) be the binding-names function of a given NTS. From now on we restrict the attention to the NTS of [23], and the definitions and results to come apply to NRTS/NRTSS over a signature \( \Sigma_{\text{NTS}} \). We require that the rules of an NRTSS only contain ground actions \( \ell \) and therefore function \( \text{bn} \) is always defined over \( NT[\ell] \). (Recall that we write \( \text{bn}(\ell) \) instead of \( \text{bn}(NT[\ell]) \) since it is clear in this context that the \( \ell \) stands for a nominal term.) The rule format that we introduce in Definition 27 relies on identifying the rules that give rise to transitions with actions \( \ell \) such that \( \text{bn}(\ell) \) is non-empty. To this end, we adapt the notion of strict stratification from [3, 14].

**Definition 26** (Partial strict stratification). Let \( \mathcal{R} \) be an NRTSS over a signature \( \Sigma_{\text{NTS}} \) and \( \text{bn} \) be a binding-names function. Let \( S \) be a partial map from pairs of ground processes and actions to ordinal numbers. \( S \) is a partial strict stratification of \( \mathcal{R} \) with respect to \( \text{bn} \) iff

(i) \( S(\varphi(t), \ell) \neq \bot \), for every rule in \( \mathcal{R} \) with conclusion \( t \rightarrow (\ell, t') \) such that \( \text{bn}(\ell) \) is non-empty and for every ground substitution \( \varphi \), and

(ii) \( S(\varphi(u_i), \ell_i) < S(\varphi(t), \ell) \) and \( S(\varphi(u_i), \ell_i) \neq \bot \), for every rule in \( \mathcal{R} \) with conclusion \( t \rightarrow (\ell, t') \) such that \( S(\varphi(t), \ell) \neq \bot \), for every premis \( u_i \rightarrow (\ell_i, u_i') \) of \( \mathcal{R} \) and for every ground substitution \( \varphi \).

We say a pair \((p, \ell)\) of ground process and action has order \( S(p, \ell) \).

The choice of \( S \) determines which rules will be considered by the rule format for NRTSSs of Definition 27 below, which guarantees that the induced transition relation satisfies alpha-conversion of residuals and, therefore, the associated transition relation together with function \( \text{bn} \) are indeed an NTS. We will intend the map \( S \) to be such that the only rules whose source and label of the conclusion have defined order are those that may take part in proof trees of transitions with some binding atom in the action.

**Definition 27** (Alpha-conversion-of-residuals format). Let \( \mathcal{R} \) be an NRTSS over a signature \( \Sigma_{\text{NTS}} \), \( \text{bn} \) be a binding-names function and \( S \) be a partial strict stratification of \( \mathcal{R} \) with respect to \( \text{bn} \). Assume that all the actions occurring in the rules of \( \mathcal{R} \) are ground. Let

\[
\frac{\{u_i \rightarrow (\ell_i, u_i') \mid i \in I\} \quad \Delta}{t \rightarrow (\ell, t') \quad \text{RU}}
\]

be a rule in \( \mathcal{R} \). Let \( D \) be the set of variables that occur in the source \( t \) of \( \text{RU} \) but do not occur in the premises \( u_i \rightarrow (\ell_i, u_i') \) with \( i \in I \), the environment \( \varphi \) or the target \( t' \) of the rule. The rule \( \text{RU} \) is in alpha-conversion-of-residuals format with respect to \( S \) (ACR format with respect to \( S \) for short) iff for each ground substitution \( \varphi \) such that \( S(\varphi(t), \ell) \neq \bot \), there exists a ground substitution \( \gamma \) such that \( \text{dom}(\gamma) \subseteq D \), and for every atom \( a \) in the set \( \text{cof}(\varphi) \) \( \forall c \supseteq \text{supp}(\varphi) \) \( (\{c \neq t \})_{\text{cof}} = \emptyset \) and for every atom \( b \in \text{bn}(\ell) \), the following hold:

(i) \( \{a \neq t'\} \cup \varphi \models \{a \neq u_i' \mid i \in I\} \),

(ii) \( \{a \neq t'\} \cup \varphi \cup \{a \neq u_i \mid i \in I\} \models \{a \neq \gamma(t)\} \), and

(iii) \( \varphi \cup \{b \neq u_i \mid i \in I \land b \in \text{bn}(\ell)\} \models \{b \neq \gamma(t)\} \).

An NRTSS \( \mathcal{R} \), together with a binding-names function \( \text{bn} \) is in ACR format iff \( \mathcal{R} \) is in equivariant format and there exists a partial strict stratification \( S \) such that all the rules in \( \mathcal{R} \) are in ACR format with respect to \( S \).
Given a transition $p \rightarrow (\ell, q)$ that unifies with the conclusion of $Ru$, the rule format ensures that any atom $a$ fresh in $(\ell, q)$ is also fresh in $p$, and also that the binding atom $b$ is fresh in $p$. We have obtained the constraints of the rule format by considering the variable flow in each node of a proof tree and the freshness relations that we want to ensure. Constraints (i) and (ii) cover the case for the freshness relation $a \# p$ and Constraint (iii) covers the case for the freshness relation $b \# p$. The purpose of substitution $\gamma$ is to ignore the variables that occur in the source of a rule but are dropped everywhere else in the rule. Constraints (i) and (ii) are not required for atoms $a$ that for sure are fresh in $p$, and this explains why the $a$ in the rule format ranges over $A \setminus \{c \in supp(t) \mid \{c \neq t\}_{af} = \emptyset\}$. For instance, take rule $ResB$ from Section 6. Condition $\{c \neq (boutA(a, b), new([c]y)), c \neq boutA(a, b)\}$ does not hold because $c \neq y$. However, for a transition $NT[new([c]p)] \rightarrow NT[(\ell, new([c]p'))]$, $c$ is fresh in $NT[new([c]p)]$ even if $c$ is not fresh in $NT[p]$.

**Theorem 28 (Rule format for NTS).** Let $R$ be an NRTSS. If $R$, together with the binding-names function $bn$, is in ACR format then the NRTS induced by $R$ and $bn$ constitute an NTS—that is, the transition relation induced by $R$ is equivariant and satisfies alpha-conversion of residuals.

**Sketch of the proof.** Given a transition $NT[\varphi(t)] \rightarrow NT[\varphi((\ell, t'))]$, we first prove the freshness relations $a \# NT[\varphi(\gamma(t))]$ and $b \# NT[\varphi(\gamma(t))]$. Both relations are proven by induction on $S(\varphi(\gamma(t)), \ell)$, and by analysing the variable flow in the rule unifying with $\varphi(t) \rightarrow \varphi((\ell, t'))$. For the first relation, we assume $a \# NT[\varphi(\gamma(t))]$, use Constraint (i) to prove that $a \# NT[\varphi(u_i')]$ for each target $u_i'$ of a premiss, apply the induction hypothesis to obtain $a \# NT[\varphi(\gamma(u_i))]$ for each source of a premiss $u_i$, and use Constraint (ii) to conclude that $a \# NT[\varphi(\gamma(t))]$. For the second relation, the induction hypothesis ensures that $b \# NT[\varphi(\gamma(u_i))]$ for each source $u_i$ of a premiss having $b$ as a binding name, and we use Constraint (iii) to conclude that $b \# NT[\varphi(\gamma(t))]$. From these two freshness relations it is straightforward to prove that $NT[\varphi(t)] \rightarrow (a b) \cdot NT[\varphi((\ell, t'))]$ and we are done.

### 6 Example of application to the early $\pi$-calculus

Consider the NRTSS $R$ for the early $\pi$-calculus [21] over a signature $\Sigma_{NTS}$ where $F$ is the set of function symbols from Example 7. Below we collect an excerpt of the rules, where $a, b, c \in A_{ec}$ and $\ell$ is a ground action:

\[
\begin{align*}
\text{in}(a, [c]x) & \quad \text{IN}\quad \text{OPEN} \quad x \rightarrow (\text{out}(a, b), y) \quad b \neq a \\
\text{sum}(x_1, x_2) & \rightarrow (\ell, y) \quad \text{SUML} \quad \ell \notin \text{bout}(a, b) \\
\text{par}(x_1, x_2) & \rightarrow (\ell, y_1, y_2) \quad \text{PAR} \quad x \rightarrow (\ell, y) \\
\text{out}(a, b, x) & \rightarrow (\text{out}(a, b), x) \quad \text{OUT} \quad \text{REP} \\
\end{align*}
\]

\[
\begin{align*}
\text{par}(x_1, x_2) & \rightarrow (\text{bout}(a, b), y_1) \quad b \neq x_2 \quad \text{PARRESL} \\
\text{par}(x_1, x_2) & \rightarrow (\text{tau}(a, b), (\text{par}(y_1, x_2))) \\
\text{rep}(x) & \rightarrow (\ell, (\text{par}(y, \text{rep}(x)))) \\
\text{par}(x_1, x_2) & \rightarrow (\text{tau}(a, b), (\text{new}(b) \cdot (\text{par}(y_1, y_2)))) \quad \text{CLOSEL} \\
\end{align*}
\]
An input process $NT\[\text{in}(a, [c]p)\]$ can perform a transition to a process $NT\[\{(cb) \cdot p\}\]$ that is obtained by substituting a channel name $b$ received through channel $a$ for channel name $c$ in $p$. In the rule In, the moderated term $(cb) \cdot x$ needs to be used in order to indicate that permutation $(cb)$ will be performed over the term substituted for variable $x$.

The rule CloseL specifies the interaction of a process such as $NT\[\text{new}([b](\text{out}(a, b, p)))\]$, which exports a private channel name $b$ through channel $a$, composed in parallel with an input process such as $NT\[\text{in}(a, [c]q)\]$ that reads through channel $a$. The private name $b$ is exported and the resulting process $NT\[\text{new}([b](\text{par}(p, (cb) \cdot q)))\]$ is the parallel composition of processes $p$ and $q$ where atom $b$ is restricted. For illustration, consider the raw terms $t \equiv \text{new}([b](\text{out}(a, b, p)))$ and $t' \equiv (\text{bout}(A, b, p))$. The transition $NT[t] \rightarrow NT[t']$ is provable in $R$ by the following proof tree:

$$\frac{\text{out}(a, b, p) \rightarrow NT\[(\text{out}(a, b, p))\]}{NT\[\text{new}([b](\text{out}(a, b, p)))\]} \rightarrow NT\[(\text{bout}(A, b, p))\]$$

Notice that the nodes of the proof tree above are labelled by transitions involving nominal terms. Therefore, if we were to start with the raw term $q \equiv \text{new}([c](\text{out}(a, c, p)))$—which is alpha-equivalent to $t$—then the transition $NT[q] \rightarrow NT[t]$ would have the same proof tree as above, since $NT[t]$ and $NT[q]$ are the same nominal term. This contrasts with the related work [7, 9], which considers raw terms in the model of computation and instruments alpha-conversion explicitly in the specification system.

We use the rule format of Definition 27 to show that $R$, together with equivariant function $\text{bn}(\ell) = \{b \mid \ell = \text{bout}(A, b)\}$ specifies an NTS. We consider the following partial strict stratification:

$$S(\text{out}(a, b, p), \text{out}(A, b)) = 0$$
$$S(\text{par}(p, q), \ell) = 1 + \max\{S(p, \ell), S(q, \ell)\} \quad \ell \in \{\text{bout}(A, b), \text{out}(A, b)\}$$
$$S(\text{sum}(p, q), \ell) = 1 + \max\{S(p, \ell), S(q, \ell)\} \quad \ell \in \{\text{bout}(A, b), \text{out}(A, b)\}$$
$$S(\text{rep}(p), \ell) = 1 + S(p, \ell) \quad \ell \in \{\text{bout}(A, b), \text{out}(A, b)\}$$
$$S(\text{new}([c]p), \ell) = 1 + S(p, \ell) \quad \ell \in \{\text{bout}(A, b), \text{out}(A, b)\} \text{ and } c \notin \{a, b\}$$
$$S(\text{new}([b]p), \text{bout}(A, b)) = 1 + S(p, \text{out}(A, b))$$
$$S(p, \ell) = \bot \quad \text{o.w.}$$

We check that $R$ is in ACR format as follows. The only rules in $R$ whose sources and actions unify with pairs of processes and actions that have defined order are Out, Open and ParResL, and the instance of rule ParL where $\ell = \text{out}(A, b)$, and the instances of rules SumL, Rep and Res where $\ell \in \{\text{bout}(A, b), \text{out}(A, b)\}$ (and the corresponding instances of the symmetric versions ParResR, ParR and SumR, which are omitted in the excerpt). We will only check the ACR-format for rules Out, SumL and Open.

For rule Out, we have an empty set of premisses and the set $D$ of atoms that are in $\text{supp}(\text{out}(a, b, x))$ but are not in $\text{supp}(\text{out}(A, b, x))$ is empty. Therefore we can do away with substitution $\gamma$. There is no atom $a$ such that $\{a \notin \text{out}(a, b, x)\} = \emptyset_{\text{nf}}$ and the set $\text{bn}(\text{out}(A, b))$ is empty. We only need to check that for every atom $c$, $\{c \notin \text{out}(A, b, x)\} \vdash \{c \notin \text{out}(a, b, x)\}$. For atoms $c \in \text{supp}(\text{out}(A, b, x))$ the obligation of the rule format vacuously holds, and therefore it is enough to pick an atom $c$ fresh in the rule and check that $\{c \notin \text{out}(A, b, x)\} \vdash \{c \notin \text{out}(a, b, x)\}$, which is straightforward.

For rule SumL, we first check the instance where $\ell = \text{bout}(A, b)$. We have premiss $x_1 \rightarrow (\text{bout}(A, b), y_1)$ and the set $D$ contains $x_2$. We pick $\gamma$ such that $\gamma(x_2) = \text{null}$. There is no atom $a$ such that $\{(a \notin \text{sum}(x_1, x_2))\}_{\text{nf}} = \emptyset$ and the set $\text{bn}(\text{bout}(A, b))$ contains atom.
b. Again, it is enough to pick atom \( c \) fresh in the rule and check that

\[
\begin{align*}
\{ c \not\approx (\text{bout}(a,b),y_1) \} \vdash \{ c \not\approx (\text{bout}(a,b), y_1) \} \quad \text{and} \\
\{ c \not\approx (\text{bout}(a,b), y_1), c \not\approx x_1 \} \vdash \{ c \not\approx \gamma(\text{sum}(x_1,x_2)) \} \quad \text{and} \\
\{ b \not\approx x_1 \} \vdash \{ b \not\approx \gamma(\text{sum}(x_1,x_2)) \},
\end{align*}
\]

which holds since \( \gamma(\text{sum}(x_1,x_2)) = \text{sum}(x_1,\text{null}) \) and \( b \not\approx \text{null} \) reduces to the empty set.

Now we check the instance where \( \ell = \text{out}(a,b) \). We have premises \( x_1 \rightarrow (\text{out}(a,b),y_1) \) and the set \( D \) and the substitution \( \gamma \) are the same as before. There is no atom \( a \) such that \( \left\langle \{ a \not\approx \text{sum}(x_1,x_2) \} \right\rangle_{nf} = \emptyset \) and the set \( \text{bu}(\text{out}(a,b)) \) is empty. Again, it is enough to pick atom \( c \) fresh in the rule and check that \( \{ c \not\approx \langle \text{out}(a,b), y_1 \rangle \} \vdash \{ c \not\approx \langle \text{out}(a,b), y_1 \rangle \} \) and \( \{ c \not\approx \langle \text{out}(a,b), y_1 \rangle, c \not\approx x_1 \} \vdash \{ c \not\approx \gamma(\text{sum}(x_1,x_2)) \} \), which holds as before.

For rule \text{OPEN} the set \( D \) is empty and \( \left\langle \{ b \not\approx \text{new}(\langle y \rangle) \} \right\rangle_{nf} = \emptyset \). It is enough to pick atom \( c \) fresh in the rule (and therefore different from \( b \)) and check that

\[
\begin{align*}
\{ c \not\approx (\text{bout}(a,b), y), b \not\approx a \} \vdash \{ c \not\approx (\text{bout}(a,b), y) \} \quad \text{and} \\
\{ c \not\approx (\text{bout}(a,b), y), b \not\approx a, c \not\approx x \} \vdash \{ c \not\approx \text{new}(\langle y \rangle x) \} \quad \text{and} \\
\{ b \not\approx x, b \not\approx a \} \vdash \{ b \not\approx \text{new}(\langle y \rangle x) \},
\end{align*}
\]

which holds because \( b \not\approx \text{new}(\langle y \rangle x) \) reduces to the empty set.

Atoms \( a, b \) and \( c \) in the specification of \( R \) range over \( \mathcal{A}_{ch} \), and thus \( R \) is in equivariant format. Therefore \( R \) is in ARC format. By Theorem 28 the NRTS induced by \( R \), together with function \( \text{bn} \), constitute an NTS of Definition 3.

7 Conclusions and future work

The work we have presented in this paper stems from the recently proposed Nominal SOS (NoSOS) framework [7] and from earlier proposals for nominal logic in [8, 15, 30]. It is by no means the only approach studied so far in the literature that aims at a uniform treatment of binders and names in programming and specification languages. Other existing approaches that accommodate variables and binders within the SOS framework are those proposed by Fokkink and Verhoef in [13], by Middelburg in [19, 20], by Bernstein in [5], by Ziegler, Miller and Palamidessi in [31] and by Fiore and Staton in [10] (originally, by Fiore and Turi in [11]). The aim of all of the above-mentioned frameworks is to establish sufficient syntactic conditions guaranteeing the validity of a semantic result (congruence in the case of [5, 10, 19, 31] and conservativity in the case of [13, 20]). In addition, Gabbay and Mathijssen present a nominal axiomatisation of the \( \lambda \)-calculus in [16]. None of these approaches addresses equivariance nor the property of alpha-conversion of residuals in [23].

Our current proposal aims at following closely the spirit of the seminal work on nominal techniques by Gabbay, Pitts and their co-workers, and paves the way for the development of results on rule formats akin to those presented in the aforementioned references. Amongst those, we consider the development of a congruence format for the notion of bisimilarity presented in [23, Def. 2] to be of particular interest. The logical characterisation of bisimilarity given in [23] opens the intriguing possibility of employing the divide-and-congruence approach from [12] to obtain an elegant congruence format and a compositional proof system for the logic.

In the NTSs of Parrow et al. [23], scope opening is modelled by the property of alpha-conversion of residuals. We are currently exploring an alternative in which scope opening is encoded by a residual abstraction of sort \([ch](\mathcal{A} \times \mathcal{P}R)\). We have developed mutual, one-to-one translations between the NTSs and the NRTSs with residual abstractions. The generality of
our NRTSs also allows for neat specifications of variants of the \( \pi \)-calculus such as Sangiorgi’s internal \( \pi \)-calculus [28].

Developing rule formats for SOS is always the result of a trade-off between ease of application and generality. Our rule format for alpha-conversion of residuals in Definition 27 is no exception and might be generalised in various ways. For instance, the quantification on atom \( a \) in conditions (i) and (ii), and the use of substitution \( \gamma \) might be made more general by a finer analysis of the variable flow in a rule. Another generalisation of the rule format would consider possibly open raw actions.

Finally, we are developing rule formats for properties other than alpha-conversion of residuals. One such rule format ensures a non-dropping property for NRTSs to the effect that, in each transition, the support of a state is a subset of the support of its derivative.

References


Divergence and Unique Solution of Equations*

Adrien Durier¹, Daniel Hirschhoff², and Davide Sangiorgi³

¹ Université de Lyon, CNRS, ENS de Lyon, UCB Lyon 1, LIP, France and
Università di Bologna and INRIA, Italy
² Université de Lyon, CNRS, ENS de Lyon, UCB Lyon 1, LIP, France
³ Università di Bologna and INRIA, Italy

Abstract

We study proof techniques for bisimilarity based on unique solution of equations. We draw inspiration from a result by Roscoe in the denotational setting of CSP and for failure semantics, essentially stating that an equation (or a system of equations) whose infinite unfolding never produces a divergence has the unique-solution property. We transport this result onto the operational setting of CCS and for bisimilarity. We then exploit the operational approach to: refine the theorem, distinguishing between different forms of divergence; derive an abstract formulation of the theorems, on generic LTSs; adapt the theorems to other equivalences such as trace equivalence, and to preorders such as trace inclusion. We compare the resulting techniques to enhancements of the bisimulation proof method (the ‘up-to techniques’). Finally, we study the theorems in name-passing calculi such as the asynchronous $\pi$-calculus, and revisit the completeness proof of Milner’s encoding of the $\lambda$-calculus into the $\pi$-calculus for Lévy-Longo Trees.

1998 ACM Subject Classification F.3.2 Semantics of Programming Languages

Keywords and phrases Bisimilarity, unique solution of equations, termination, process calculi

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2017.11

1 Introduction

In this paper we study the technique of unique solution of equations for (weak) behavioural relations. We mainly focus on bisimilarity but we also consider other equivalences, such as trace equivalence, as well as preorders such as trace inclusion. Roughly, the technique consists in proving that two tuples of processes are componentwise in a given behavioural relation by establishing that they are solutions of the same system of equations.

In this work, behavioural relations, hence also bisimilarity, are meant to be weak because they abstract from internal moves of terms, as opposed to the strong relations, which make no distinctions between the internal moves and the external ones (i.e., the interactions with the environment). Weak equivalences are, practically, the most relevant ones: e.g., two equal programs may produce the same result with different numbers of evaluation steps. Further, the problems tackled in this paper only arise in the weak case.

The technique of unique solution has been proposed by Milner in the setting of CCS, and plays a prominent role in proofs of examples in his book [13]. The method is important in verification techniques and tools based on algebraic reasoning [22, 2, 10]. Not all equations have a unique solution: for instance any process trivially satisfies $X = X$. In Milner’s theorem [13], uniqueness of solutions is subject to some limitations: the equations must

* This work was supported by Labex MILYON/ANR-10-LABX-0070, and by the project ANR-16-CE25-0011 REPAS.

© Adrien Durier, Daniel Hirschhoff, and Davide Sangiorgi;
licensed under Creative Commons License CC-BY
28th International Conference on Concurrency Theory (CONCUR 2017).
Editors: Roland Meyer and Uwe Nestmann; Article No. 11; pp. 11:1–11:16
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany
be ‘strongly guarded and sequential’, that is, the variables of the equations may only be used underneath a visible prefix and preceded, in the syntax tree, only by the sum and prefix operators. This limits the expressiveness of the technique (since occurrences of other operators above the variables, such as parallel composition and restriction, in general cannot be removed), and its transport onto other languages (e.g., languages for distributed systems or higher-order languages usually do not include the sum operator, which makes the theorem essentially useless). A comparable technique, involving similar limitations, has been proposed by Hoare in his CSP book [12].

Trying to overcome such limitations, a variant of the technique, called unique solution of contractions has been proposed [25]. The technique is for behavioural equivalences; however the meaning of ‘solution’ is defined in terms of the contraction of the chosen equivalence. Contraction is, intuitively, a preorder that conveys an idea of efficiency on processes, where efficiency is measured on the number of internal actions needed to perform a certain activity. The condition for applicability of the technique is, as for Milner’s, purely syntactic: each variable in the body of an equation should be underneath a prefix. The technique has two main disadvantages: for proving an equivalence one needs also the theory of the associated contraction preorder; there may be processes for which the technique is not applicable simply because the contraction is strictly finer than the equivalence, and therefore one of the processes fails to be a solution.

In this paper we explore a different approach, inspired by results by Roscoe in CSP [21, 20], essentially stating that a guarded equation (or system of equations) whose infinite unfolding never produces a divergence has the unique-solution property. The theorem is presented, as usual in CSP, with respect to denotational semantics and failure based equivalence [5, 6]. In such a setting, where divergence is catastrophic (e.g., it is the bottom element of the domain), the theorem has an elegant and natural formulation. (Indeed, Roscoe develops a denotational model [20] in which the proof of the theorem is just a few lines.)

We draw inspiration from Roscoe’s work to formulate the counterpart of these results in the operational setting of CCS and bisimilarity. In comparison with the denotational CSP proof, the operational CCS proof is more complex. The operational setting offers however a few advantages. First, we can formulate more refined versions of the theorem, in which we distinguish between different forms of divergence. (These refinements would look less natural in the denotational and trace-based setting of CSP, where any divergence causes a process to be considered undefined.) A second and more important advantage comes as a consequence of the flexibility of the operational approach: the unique-solution theorems can be tuned to other behavioural relations (both equivalences and preorders), and to other languages.

To highlight the latter aspect, we present abstract formulations of the theorems, on a generic LTS (i.e., without reference to CCS), where the body of an equation becomes a function on the states of the LTS. The CCS theorems are instances of the abstract formulations. Similarly we can derive analogous theorems for other languages. Indeed we can do so for all languages whose constructs have an operational semantics with rules in the GSOS format [3] (assuming appropriate hypotheses, among which congruence properties). In contrast, the analogous theorems fail for languages whose constructs follow the tyft/tyxt [11] format, due to the possibility of rules with a lookahead. We also consider extensions of the theorems to name-passing calculi such as the π-calculus. The abstract version of our main unique-solution theorem has been formalised using the Coq proof assistant [8].

Today, for concrete proofs of bisimilarity results, the bisimulation proof method is predominant, also thanks to enhancements of the method provided by the so called ‘up-to techniques’ [19]. Powerful enhancements are ‘up to context’, whereby in the derivatives of
two terms a common context can be erased, ‘up to expansion’, whereby two derivatives can be rewritten using the expansion preorder, and ‘up to transitivity’, whereby the matching between two derivatives is made with respect to the transitive closure of the candidate relation (rather than the relation alone). Different enhancements can sometimes be combined, though care is needed to preserve soundness. One of the most powerful combinations is Pous ‘up to transitivity and context’ technique, which relies on a termination hypothesis. This technique generalises ‘up to expansion’ and combines it with ‘up to context’ and ‘up to transitivity’. We show that, under an additional side condition, our techniques are at least as powerful as this up-to technique: any up-to relation can be turned into a system of equations of the same size as the up-to relation and that satisfies the hypothesis of our theorems.

An important difference between unique solution of equations and up-to techniques arises in the (asynchronous) π-calculus. In this setting, forms of bisimulation enhancements that involve ‘up to context’ require closure of the candidate relation under substitutions (or instantiation of the parameters of an abstraction with arbitrary values). It is an open problem whether this closure is necessary in the asynchronous π-calculus, where bisimilarity is closed under substitutions. Our unique-solution techniques are strongly reminiscent of up to context techniques (the body of an equation acts like a context that is erased in a proof using ‘up to context’); yet, surprisingly, no closure under substitutions is required.

As an example of application of our techniques in the π-calculus we revisit the completeness part of the proof of full abstraction for the encoding of the λ-calculus into the π-calculus [24, 27] with respect to Levy-Longo Trees (LLT). The proof in [24, 27] uses ‘up to expansion and context’. Such up-to techniques seem to be essential: without them, it would be hard even to define the bisimulation candidate. For our proof using unique-solution, there is one equation for each node of a given LT, describing the shape of such node.

Outline of the paper. Section 2 provides background about CCS and behavioural relations. We formulate our main results for CCS in Section 3, and generalise them in an abstract setting in Section 4. Section 5 shows how our results can be applied to the π-calculus.

2 Background

CCS. We assume an infinite set of names a, b, . . . and a set of constant identifiers (or simply constants) to write recursively defined processes. The special symbol τ does not occur in the names and in the constants. We recall the grammar of CCS:

\[
P := P_1 | P_2 | \Sigma_{i \in I} \mu_i, P_i | \nu a P | K \quad \mu := a | \overline{a} | \tau
\]

where I is a countable indexing set. We write 0 when I is empty, and P + Q for binary sums. Each constant K has a definition K \triangleleft P. We sometimes omit trailing 0, e.g., writing a | b for a.0 | b.0. The operational semantics is given by means of an LTS, and is given in Figure 1 (the symmetric versions of the rules parL and comL have been omitted).

Some standard notations for transitions: ⇒ is the reflexive and transitive closure of \(\to\), and \(\overset{\mu}{\Rightarrow}\) is \(\Rightarrow \overset{\mu}{\Rightarrow}\) (the composition of the three relations). Moreover, \(P \overset{\mu}{\Rightarrow} P'\) holds if \(P \overset{\mu}{\Rightarrow} P'\) or (\(\mu = \tau\) and \(P = P'\)); similarly \(P \overset{\mu}{\Rightarrow} P'\) holds if \(P \overset{\mu}{\Rightarrow} P'\) or (\(\mu = \tau\) and \(P = P'\)).

Letters \(R, S\) range over relations. We use the infix notation for relations, e.g., \(P R Q\) means that \((P, Q) \in R\), and denote as \(RS\) the composition of \(R\) and \(S\). A relation terminates if there is no infinite sequence \(P_1 R P_2 R \ldots\). We use a tilde to denote a tuple, with countably many elements; thus the tuple may also be infinite. All notations are extended to tuples...
componentwise; e.g., \( \tilde{P} R \tilde{Q} \) means that \( P_i R Q_i \), for each component \( i \) of the tuples \( \tilde{P} \) and \( \tilde{Q} \). We use \( \text{def} \) for abbreviations; in contrast, \( \triangleq \) is used for the definition of constants, and \( = \) for syntactic equality and for equations. We focus on weak behavioural equivalences, which abstract from the number of internal steps performed.

\[ \begin{align*}
\frac{\text{sum}}{\sum_{i \in I} \mu_i, P_i \xrightarrow{\mu_i} P} & \quad \frac{\text{parL}}{P \parallel Q \xrightarrow{\mu} P' | Q} & \frac{\text{comL}}{P \xrightarrow{\alpha} P', Q \xrightarrow{\pi} Q'} \\
\frac{\text{res}}{\nu a \ P \xrightarrow{\nu a P'} \mu \neq a, \pi} & \quad \text{const} & \frac{P \xrightarrow{\mu} P', \ K \xrightarrow{\mu'} P'}{\text{if } K \triangleq P}
\end{align*} \]

**Definition 1** (Bisimulation). A relation \( R \) is a bisimulation if, whenever \( P R Q \), we have:

1. \( P \xrightarrow{\tau} P' \) implies that there is \( Q' \) such that \( Q \xrightarrow{\tau} Q' \) and \( P' R Q' \);
2. the converse, on the actions from \( Q \).

\( P \) and \( Q \) are bisimilar, written \( P \approx Q \), if \( P R Q \) for some bisimulation \( R \).

**Systems of equations.** Unique solution of equations [13] intuitively says that if a context \( C \) obeys certain conditions, then all processes \( P \) that satisfy the equation \( P \approx C[P] \) are bisimilar with each other.

We need variables to write equations. We use capital letters \( X, Y, Z \) for these variables and call them equation variables. The body of an equation is a CCS expression possibly containing equation variables. We use \( E, E' \ldots \) to range over equation expressions; these are process expressions that may contain occurrences of variables; that is, the grammar for processes is extended with a production for variables.

**Definition 2.** Assume that, for each \( i \) of a countable indexing set \( I \), we have a variable \( X_i \), and an expression \( E_i \), possibly containing some variables. Then \( \{ X_i = E_i \}_{i \in I} \) (sometimes written \( \hat{X} = \hat{E} \)) is a system of equations. (There is one equation for each variable \( X_i \).)

\( E[\tilde{P}] \) is the process resulting from \( E \) by replacing each variable \( X_i \) with the process \( P_i \), assuming \( \tilde{P} \) and \( \hat{X} \) have the same length. (This is syntactic replacement.) The components of \( \tilde{P} \) need not be different from each other, while this must hold for the variables \( \hat{X} \).

**Definition 3.** Suppose \( \{ X_i = E_i \}_{i \in I} \) is a system of equations. We say that:

- \( \tilde{P} \) is a solution of the system of equations for \( \approx \) if for each \( i \) it holds that \( P_i \approx E_i[\tilde{P}] \).
- The system has a unique solution for \( \approx \) if whenever \( \tilde{P} \) and \( \tilde{Q} \) are both solutions for \( \approx \), then \( \tilde{P} \approx \tilde{Q} \).

For instance, the system \( X_1 = a, X_2, X_2 = b \). \( X_1 \) has a unique solution, whereas the equations \( X = X \), or \( \tau. X \), or \( X = a \mid X \) do not.

A system of equations is guarded (resp. strongly guarded) if each occurrence of a variable in the body of an equation is underneath a prefix (resp. a visible prefix, i.e., different from \( \tau \)).

**Divergences.** We consider the standard notion of divergence:

**Definition 4** (Divergence). A process \( P \) diverges if it can perform an infinite sequence of internal moves, possibly after some visible ones; i.e., there are processes \( P_i, i \geq 0 \), and some \( n \), such that \( P = P_0 \xrightarrow{\mu_0} P_1 \xrightarrow{\mu_1} P_2 \xrightarrow{\mu_2} \ldots \) and for all \( i > n, \mu_i = \tau \). We call a divergence of \( P \) the sequence of transitions \( \{ P_i \xrightarrow{\mu} P_{i+1} \}_{i \geq 0} \).
Example 5. The process $L \overset{a}{\rightarrow} \nu a (L \mid \pi)$ diverges, since $L \overset{a}{\rightarrow} \nu a (L \mid \pi)$, and (leaving aside 0 and useless restrictions) $\nu a (L \mid \pi)$ has a $\tau$ transition onto itself.

3 Main Results

3.1 Divergences and Unique Solution

This section is devoted to our main results for bisimilarity, in the case of CCS. We need to reason with the unfoldings of the given equation $X = E$: we define the $n$-th unfolding of $E$ to be $E^n$; thus $E^1$ is defined as $E$, $E^2$ as $E[E]$, and $E^{n+1}$ as $E^n[E]$. The infinite unfolding represents the simplest and most intuitive solution to the equation. In the CCS grammar, such a solution is obtained by turning the equation into a constant definition, namely the constant $K_E$ with $K_E \overset{\triangle}{=} E[K_E]$. We call $K_E$ the syntactic solution of the equation.

For a system of equations $\tilde{X} = \tilde{E}[\tilde{X}]$, the unfoldings are defined accordingly (where $E_i$ replaces $X_i$ in the unfolding), and the syntactic solutions are defined to be the set of mutually recursive constants $\{K_{E,i} \overset{\triangle}{=} E_i[K_{E,i}]\}$.

Theorem 6 (Unique solution). A guarded system of equations whose syntactic solutions do not diverge has a unique solution for $\approx$.

We explain the schema of the proof, considering, for simplicity, a single equation $X = E$. We take a solution $P$ of the equation and a transition $P \overset{\mu}{\rightarrow} P'$. The goal is to find an $n$ such that $E^n[P]$ can match this transition without the need of $P$; i.e., there is $E'$ with $E^n[P] \overset{\mu}{\rightarrow} E'[P]$, and for any process $Q$ also $E^n[Q] \overset{\mu}{\rightarrow} E'[Q]$ holds.

We look for this $n$ incrementally. If the matching transition $E^n[P] \overset{\mu}{\rightarrow} P_m$ (recall the $P$ is solution) involves some transitions of $P$, then $E^n[P]$ does not work. We then consider a matching transition emanating from $E^{n+1}[P]$, which starts with the transitions in $E^n[P] \overset{\mu}{\rightarrow} P_m$ that do not involve $P$. We observe that there are at least $m$ of these, because $P$ is underneath at least $m$ prefixes in $E^n[P]$.

This procedure necessarily stops: otherwise, we could build an infinite sequence of transitions involving only the unfoldings of $E$, and with at most one visible transition: this would yield a divergence in the syntactic solution of $E$. With this construction at hand, given another solution $Q$ of the equation, we construct a bisimulation containing the pair $(P, Q)$.

3.2 Innocuous Divergences

In the remainder of the section we refine Theorem 6 by taking into account only certain forms of divergence. To introduce the idea, consider the equation $X = a \cdot X \mid K$, for $K \overset{\triangle}{=} \tau \cdot K$: the divergences induced by $K$ do not prevent uniqueness of the solution, as any solution $P$ necessarily satisfies $P \approx a \cdot P$. Indeed the variable of the equation is strongly guarded and a visible action has to be produced before accessing the variable. These divergences are not dangerous because they do not percolate through the infinite unfolding of the equation; in other words, a finite unfolding may produce the same divergence, therefore it is not necessary to go to the infinite unfolding to diverge. We call such divergences innocuous. Formally, these divergences are derived by applying only a finite number of times rule const of the LTS (see Figure 1) to the constant that represents the syntactic solution of the equation.

Definition 7 (Innocuous divergence). Consider a guarded system of equations $\tilde{X} = \tilde{E}$ and its syntactic solutions $\tilde{K}_{\tilde{E}}$. A divergence of $\tilde{K}_{\tilde{E},i}$ (for some $i$) is called innocuous when
summing up all usages of rule const with one of the $K_{E,j}$s (including $j = i$) in all derivation proofs of the transitions belonging to the divergence, we obtain a finite number.

**Theorem 8** (Unique solution with innocuous divergences). Let $\tilde{X} = \tilde{E}$ be a system of guarded equations, and $\tilde{K}_E$ be its syntactic solutions. If all divergences of any $\tilde{K}_{E,j}$ are innocuous, then $\tilde{E}$ has a unique solution for $\approx$.

**Remark.** The conditions for unique solution in Theorems 6 and 8 are a mixture of syntactic (guardedness) and semantic (divergence-free) conditions. A purely semantic condition can be used if rule const of Figure 1 is modified so that the unfolding of a constant yields a $\tau$-transition:

$$\frac{}{K \xrightarrow{\tau} P} \quad \text{if } K \subseteq P$$

Thus in the theorems the condition that the equations are guarded could be dropped. The resulting theorems would actually be more powerful because they would accept equations not all of which are guarded: it is sufficient that each equation has a finite unfolding that is guarded. For instance the system of equations $X = b \mid Y$, $Y = a$. $X$ would be accepted, although the first equation is not guarded.

The next lemma states a condition to ensure that all divergences produced by a system of equations are innocuous. This condition will be sufficient in all examples in the paper.

**Lemma 9.** In a system of equations $\tilde{X} = \tilde{E}$, suppose for each $i$ there is $n_i$ such that in $E_i^n$, each variable is underneath a visible prefix (say, $a$ or $\overline{a}$) whose complementary prefix (either $\overline{a}$ or $a$) does not appear in any equation. Then the system has only innocuous divergences.

### 3.3 An example: lazy and eager servers

We now show an example of application of our technique, taken from [25]. The example also illustrates the relative strengths of the two unique solution theorems (Theorems 6 and 8).

For the sake of readability, we use a version of CCS with value passing; this could be translated into pure CCS [13]. In a value-passing calculus, $a(x)$. $P$ is an input at $a$ in which $x$ is the placeholder for the value received, whereas $\overline{a}(n)$. $P$ is an output at $a$ of the value $n$; and $A(n)$ is a parametrised constant. This example consists of two implementations of a server; this server, when interrogated by clients at a channel $c$, should start a certain interaction protocol with the client, after consulting an auxiliary server $A$ at $a$.

We consider the two following implementations of this server: the first one, $L$, is ‘lazy’, and consults $A$ only after a request from a client has been received. In contrast, the other one, $E$, is ‘eager’, and consults $A$ beforehand, so then to be ready in answering a client:

$$
L \triangleq c(z). a(x). (L \mid R(x, z)) \quad \quad E \triangleq a(x). c(z). (E \mid R(x, z))
$$

$$A(n) \triangleq \pi(n). A(n + 1)$$

Here $R(x, z)$ represents the interaction protocol that is started with a client, and can be any process. It may use the values $x$ and $z$ (obtained from the client and the auxiliary server $A$); the interactions produced may indeed depend on the values $x$ and $z$. We assume for now that $R(x, z)$ may not use channel $c$ and $a$; that is, the interaction protocol that has been spawned need not come back to the main server or to the auxiliary server. Moreover we assume $R$ may not diverge. We want to prove that the two servers, when composed with $A$, yield bisimilar processes. We thus define $LS(n) \triangleq \nu a (A(n) \mid L)$ and $ES(n) \triangleq \nu a (A(n) \mid E)$. A proof that $LS(n) \approx ES(n)$ using the plain bisimulation proof method would be long and
tedious, due to the differences between the lazy and the eager server, and to the fact that \( R \) is nearly an arbitrary process.

For a proof using our technique, the equations are:  
\[
\{ X_n = c(z). (X_{n+1} | R(n, z)) \}_n.
\]

The proofs that the two servers are solutions can be carried out using a few algebraic laws: expansion law, structural laws for parallel composition and restriction, one \( \tau \)-law. To apply Theorem 6, we also have to check that the equations may not produce divergences. This check is straightforward, as no silent move may be produced by interactions along \( c \), and any two internal communications at \( a \) are separated by a visible input at \( c \). Moreover, by assumption, the protocol \( R \) does not produce internal divergences.

If however the hypothesis that \( R \) may not diverge is lifted, then Theorem 6 is not applicable anymore, and divergences are possible. However, such divergences are innocuous: the equation need not be unfolded an infinite number of times for the divergence to occur. We can therefore still prove the result, by appealing to the more powerful Theorem 8.

### 3.4 Comparison with other techniques

Milner’s syntactic condition for unique solution of equations essentially allows only equations in which variables are underneath prefixes and sums. The technique is not complete [25]; for instance it cannot handle the server example of Section 3.3.

The technique of ‘unique solution of contractions’ [25] relies on the theory of an auxiliary preorder (contraction), needed to establish the meaning of ‘solution’; and the soundness theorems in [25] use a purely syntactic condition (guarded variables). In contrast, our techniques with equations do not rely on auxiliary relations and their theory, but the soundness theorems use a semantic condition (divergence). The two techniques are incomparable.

Considering the server example of Section 3.3, the contraction technique is capable of handling also the case in which the protocol \( R \) is freely allowed to make calls back to the main server, including the possibility that, in doing this, infinitely many copies of \( R \) are spawned. This possibility is disallowed for us, as it would correspond to a non-innocuous divergence. On the other hand, when using contraction, a solution is evaluated with respect to the contraction preorder, that conveys an idea of efficiency (measured against the number of silent transitions performed). Thus, while two bisimilar processes are solutions of exactly the same set of equations, they need not be solutions of the same contractions. For instance, we can use our techniques to prove that processes \( K \triangleq \tau \cdot a \cdot K \) and \( H \triangleq a \cdot H \) are bisimilar because solutions of the equation \( X = a \cdot X \); in contrast, only \( H \) is a solution of the corresponding contraction.

**Up-to techniques.** We compare our unique-solution techniques with one of the most powerful forms of enhancement of the bisimulation proof method, namely Pous ‘up to transitivity and context’ technique [17]. That technique allows us to use ‘up to weak bisimilarity’, ‘up to transitivity’, and ‘up to context’ techniques together. While ‘up to weak bisimilarity’ and ‘up to transitivity’ are known to be unsound techniques [19], here they are combined at the price of a termination hypothesis over a ‘control relation’, below written \( \succ \).

We write \( C(R) \) for the context closure of a relation \( R \) (the set of all pairs \( (C[\hat{P}], C[\hat{Q}]) \)) with \( \hat{P} \sim R \sim \hat{Q} \)). Moreover, \( R^\succ \) stands for \( \approx \cup C(R) \)), and \( R^{\succ +} \) for the transitive closure of \( R \).

**Definition 10.** Let \( \succ \) be a relation that is transitive, closed under contexts, and such that \( \succ \) (\( \sim \)\+) terminates. A relation \( R \) is a bisimulation up to \( \succ \) and context if, whenever \( P \sim R \sim Q \):

1. if \( P \sim R \sim P' \) then \( Q \sim R \sim Q' \) for some \( Q' \) with \( P' \sim Q' \sim (\succ \cap R)^+ \approx Q' \);
2. the converse, on the transitions from \( Q \).
If $\mathcal{R}$ is a relation then we can also view $\mathcal{R}$ as an ordered sequence of pairs (e.g., assuming some lexicographical ordering). Then $\mathcal{R}_i$ indicates the tuple obtained by projecting the pairs in $\mathcal{R}$ on the $i$-th component ($i = 1, 2$). The size of a relation is the number of pairs it contains. The size of a system of equations is the number of equations it consists of.

**Theorem 11** (Completeness with respect to up-to techniques). Suppose $\mathcal{R}$ is a bisimulation up to $\succ$ and context. Then there is a guarded system of equations, of the same size as $\mathcal{R}$, with only innocuous divergences, admitting $\mathcal{R}_1$ and $\mathcal{R}_2$ as solutions.

In the proof, the equations are defined by exploiting the expansion law in CCS [13]. The proof then involves a rather delicate analysis in which the termination hypothesis is used to prove that the syntactic solutions of this system have only innocuous divergences (they may indeed have divergences). We may then apply Theorem 8 (rather than Theorem 6).

**Remark.** In [17], the transitive closure $(\succ \cap \mathcal{R})^+$ in Definition 10 is actually a reflexive and transitive closure. We do not know if relaxing this technical condition breaks Theorem 11.

### 4 Abstract Formulation

#### 4.1 Abstract LTS and Operators

In this section we propose generalisations of the unique-solution theorems. For this we introduce abstract formulations of them, which are meant to highlight their essential ingredients. When instantiated to the specific case of CCS, such abstract formulations yield the theorems in Section 3. The proofs are adapted from those of the corresponding theorems in Section 3. The results of this section, up to Theorem 15, have been formalised in Coq [8].

The abstract formulation is stated on a generic LTS, that is, a triple $\mathcal{T} = (S, \Lambda, \to)$ where: $S$ is the set of states; $\Lambda$ the set of action labels, containing the special label $\tau$ accounting for silent actions; $\to$ is the transition relation. As usual, we write $s_1 \xrightarrow{\mu} s_2$ when $(s_1, \mu, s_2) \in \to$. The definition of weak bisimilarity $\approx$ is as in Section 2. We omit explicit reference to $\mathcal{T}$ when there is no ambiguity.

We reason about state functions, i.e., functions from $S$ to $S$, and use $f, f', g$ to range over them. We recall that $(f \circ g)(x) = f(g(x))$ for all $x$. The CCS processes of Section 2 correspond here to the states of an LTS; and CCS contexts correspond to state functions.

**Definition 12** (Autonomy). For state functions $f, f'$ we say that there is an autonomous $\mu$-transition from $f$ to $f'$, written $f \xrightarrow{\mu} f'$, if for all states $x$ it holds that $f(x) \xrightarrow{\mu} f'(x)$.

Likewise, given a set $\mathcal{F}$ of state functions and $f \in \mathcal{F}$, we say that a transition $f(x) \xrightarrow{\mu} y$ is autonomous on $\mathcal{F}$ if, for some $f' \in \mathcal{F}$ we have $f \xrightarrow{\mu} f'$ and $y = f'(x)$. Moreover, we say that function $f$ is autonomous on $\mathcal{F}$ if all the transitions emanating from $f$ (that is, all transitions of the form $f(x) \xrightarrow{\mu} y$, for some $x, \mu, y$) are autonomous on $\mathcal{F}$.

When $\mathcal{F}$ is clear, we omit it, and we simply say that a function is autonomous.

Thus, $f$ is autonomous on $\mathcal{F}$ if, for some indexing set $I$, there are $\mu_i$ and $f_i \in \mathcal{F}$ such that for all $x$ we have $f(x) \xrightarrow{\mu_i} f_i(x)$, for each $i$; the set of all transitions emanating from $f(x)$ is precisely $\bigcup_i \{f(x) \xrightarrow{\mu_i} f_i(x)\}$. Autonomous functions correspond to CCS guarded contexts, which do not need their process argument to perform the first transition. We now formulate conditions under which, intuitively, a state function behaves like a CCS context. Functions satisfying these conditions are called operators.

**Definition 13** (Set of operators). Consider an LTS $\mathcal{T}$, and a set $\mathcal{O}$ of functions from $S$ to $S$. We say that $\mathcal{O}$ is a set of operators on $\mathcal{T}$ if the following conditions hold:
1. \( \mathcal{O} \) contains the identity function;
2. \( \mathcal{O} \) is closed by composition (that is, \( f \circ g \in \mathcal{O} \) whenever \( f, g \in \mathcal{O} \));
3. composition preserves autonomy (i.e., if \( g \) is autonomous on \( \mathcal{O} \), then so is \( f \circ g \));
4. all functions in \( \mathcal{O} \) respect \( \approx \), i.e., \( x \approx y \) and \( f \in \mathcal{O} \) imply \( f(x) \approx f(y) \).

A ‘symmetric variant’ of clause 3 always holds: if \( f \) is autonomous, then so is \( f \circ g \). The autonomous transitions of a set of operators yield an LTS whose states are the operators themselves. Such transitions are of the form \( f \xrightarrow{\mu} g \). Where the underlying set \( \mathcal{O} \) of operators is clear, we simply call a function belonging to \( \mathcal{O} \) an operator.

We use state functions to express equations, such as \( X = f(X) \). We thus look at conditions under which such an equation has a unique solution (again, the generalisation to a system of equations is easy, using \( n \)-ary functions).

Thinking of functions as equation expressions, to formulate our abstract theory about unique solution of equations, we have to define the divergences of finite and infinite unfoldings of state functions. The \( n \)th unfolding of \( f \) (for \( n \geq 1 \)), \( f^n \), is the function obtained by \( n \) applications of \( f \). An operator is well-behaved if there is \( n \) with \( f^n \) autonomous (the well-behaved operators correspond, in CCS, to equations some finite unfolding of which yields a guarded expression). We also have to reason about the infinite unfolding of an equation \( X = f(X) \). For this, given a set \( \mathcal{O} \) of operators, we consider the infinite terms obtained by infinite compositions of operators in \( \mathcal{O} \), that is, the set coinductively defined by the grammar:

\[
F := f \circ F \quad \text{where} \quad f \in \mathcal{O} \quad (\text{i.e.,} \ f \text{ is a metavariable for the elements in} \ \mathcal{O}).
\]

(We do not need finite compositions, as \( \mathcal{O} \) itself is closed under finite compositions.) In particular, we write \( f^\infty \) for the infinite term \( f \circ f \circ f \circ \ldots \).

We define the autonomous transitions for such infinite terms using the following rules:

\[
\begin{align*}
g \xrightarrow{\mu} g' & \quad \text{\( g \) autonomous} \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad 
\end{align*}
\]

Intuitively a term is ‘unfolded’, with the second rule, until an autonomous function is uncovered, and then transitions are computed using the first rule (we disallow unnecessary unfoldings; these would complicate our abstract theorems, by adding duplicate transitions, since the transitions of \( g \circ f \) duplicate those of \( g \) when \( g \) is autonomous). An infinite term has no transitions if none of its finite unfoldings ever yields an autonomous function. This situation does not arise for terms of the form \( f^\infty \) or \( g \circ f^\infty \), where \( f \) is well-behaved, which are the terms we are interested in. Note that no infinite term belongs to a set of operators.

Definition 14 (Operators and divergences). Let \( f, f', f_1 \) be operators in a set \( \mathcal{O} \) of operators, and consider the LTS induced by the autonomous transitions of operators in \( \mathcal{O} \). A sequence of transitions \( f_1 \xrightarrow{\mu_1} f_2 \xrightarrow{\mu_2} f_3 \ldots \) is a divergence if for some \( n \geq 1 \) we have \( \mu_i = \tau \) whenever \( i \geq n \). We also say that \( f_1 \) diverges. We apply these notations and terminology also to infinite terms (such as \( f_1 \circ f_2 \circ \ldots \)), as expected.

In the remainder of the section we fix a set \( \mathcal{O} \) of operators and we only consider autonomous transitions on \( \mathcal{O} \). We now state the “abstract version” of Theorem 6 (the proof being similar).

Theorem 15 (Unique solution, abstract formulation). Let \( f \) be a well-behaved operator on \( \mathcal{O} \). If \( f^\infty \) does not diverge, then the equation \( X = f(X) \) either has no solution or has a unique solution for \( \approx \).

The equation in the statement of the theorem might have no solution at all. For example, consider the LTS \((\mathbb{N}, \{a\}, \rightarrow)\) where for each \( n \) we have \( n+1 \xrightarrow{a} n \). The function \( f \) with
11:10 Divergence and Unique Solution of Equations

\(f(n) = n + 1\) is an operator of the set \(\mathcal{O} = \{f^n\}_{n \in \mathbb{N}}\) (with \(f^0 = \text{Id}\), the identity function). The function \(f\) is autonomous because, for all \(n\), the only transition of \(f(n)\) is \(\xrightarrow{a} n\) (this transition is autonomous because \(f \xrightarrow{a} \text{Id}\)). A fixpoint of \(f\) would be an element \(x\) with \(x \xrightarrow{a} x\), and there is no such \(x\) in the LTS.

Theorem 15 can be refined along the lines of Theorem 8. For this, we have to relate the divergences of any \(f^n\) (for \(n \geq 1\)) to divergences of \(f^\infty\), in order to distinguish between innocuous and non-innocuous divergences. To build a divergence of \(f^\infty\) from a divergence of \(f^n\), for \(n \geq 1\), we need to reason up to (finite) unfoldings of \(f\): thus we set \(\approx\) to be the symmetric reflexive transitive closure of the relation that relates \(g\) and \(g'\) whenever \(g = g' \circ f\).

\[\text{Lemma 16. Consider an autonomous operator } f \text{ on } \mathcal{O} \text{ and a divergence of } f^n\]

\[f^n \xrightarrow{\mu_1} f_1 \xrightarrow{\mu_2} \ldots \xrightarrow{\mu_i} f_i \xrightarrow{\tau} f_{i+1} \xrightarrow{\tau} \ldots\]

This yields a divergence of \(f^\infty\): \(f^\infty \xrightarrow{\mu_1} g_1 \circ f^\infty \xrightarrow{\mu_2} \ldots \xrightarrow{\mu_i} g_i \circ f^\infty \xrightarrow{\tau} g_{i+1} \circ f^\infty \xrightarrow{\tau} \ldots\)

such that for all \(i \geq 1\), \(g_i\) is an operator and \(g_i \approx f_i\).

Given a divergence \(\Delta\) of \(f^n\), we write \(\Delta^\infty\) to indicate the divergence of \(f^\infty\) obtained from \(\Delta\) as in Lemma 16. We call a divergence of \(f^\infty\) innocuous when it can be described in this way, that is, as a divergence \(\Delta^\infty\) obtained from a divergence \(\Delta\) of \(f^n\), for some \(n\).

\[\text{Theorem 17 (Unique solution with innocuous divergences, abstract formulation). Let } f \in \mathcal{O} \text{ be a well-behaved operator. If all divergences of } f^\infty \text{ are innocuous, then the equation } X = f(X) \text{ either has no solution or has a unique solution for } \approx.\]

### 4.2 Reasoning with other Behavioural Relations

**Trace-based Equivalences.** We can adapt the results of the previous section about bisimulation to other settings, including both preorders and non-coinductive relations. As an example, we consider trace-based relations. We write \(\approx_{tr}\) for trace equivalence, that equates two processes having the same set of (finite) traces. The definitions from previous sections are the same as for \(\approx\), replacing \(\approx\) with \(\approx_{tr}\). All theorems presented for \(\approx\) can be adapted to \(\approx_{tr}\), and the proofs are similar. As an example, Theorem 17 becomes:

\[\text{Theorem 18. Let } f \in \mathcal{O} \text{ be a well-behaved operator. If all divergences of } f^\infty \text{ are innocuous, then the equation } X = f(X) \text{ either has no solution or has a unique solution for } \approx_{tr}.\]

In contrast, the theorems fail for infinitary trace equivalence, \(\approx^\infty_{tr}\) (whereby two processes are equated if they have the same traces, including the infinite ones), for the same reason why the ‘unique solution of contraction’ technique fails in this case [25]. As a counterexample, we consider equation \(X = a + a \cdot X\), whose syntactic solution has no divergences. The process \(P \triangleq \sum_{n \geq 0} a^n\) is a solution, yet it is not \(\approx^\infty_{tr}\)-equivalent to the syntactic solution of the equation, because the syntactic solution has an infinite trace involving \(a\) transitions.

**Preorders.** We show how the theory for equivalences can be transported onto preorders. This means moving to systems of pre-equations, \(\{X_i \leq E_i\}_{i \in I}\). With preorders, our theorems have a different shape: we do not use pre-equations to reason about unique solution – we expect interesting pre-equations to have many solutions, some of which may be incomparable with each other. We rather derive theorems to prove that, in a given preorder, any solution of a pre-equation is below its syntactic solution.
Remark. The opposite direction for pre-equations, namely \( \{ X_i \geq E_i \}_{i \in I} \) is less interesting. It would mean aiming to prove that the syntactic solution is below other solutions. This is usually a trivial property for a behavioural preorder, without any hypothesis such as autonomy or non divergence (a possible exception is a preorder with infinitary observables, such as infinitary trace inclusion, where the property may fail).

We write \( \subseteq_{tr} \) for trace inclusion, \( \subseteq_{tr}^{\infty} \) for infinitary trace inclusion, and \( \leq_s \) for weak simulation. These preorders are standard from the literature [28].

In the abstract setting, the body of the pre-equations are functions. Then the theorems give us conditions under which, given a pre-equation \( X \leq f(X) \) and a behavioural preorder \( \preceq \), a solution \( r \), i.e., a state for which \( r \preceq f(r) \) holds, is below the syntactic solution \( f^{\infty} \). We present the counterpart of Theorem 17; other theorems are transported in a similar manner.

Theorem 19. Let \( f \in O \) be a well-behaved operator. If all divergences of \( f^{\infty} \) are innocuous, then, given \( \preceq \in \{ \subseteq_{tr}, \subseteq_{tr}^{\infty}, \leq_s \} \), whenever \( x \preceq f(x) \) we also have \( x \preceq f^{\infty}(x) \), for any state \( x \).

Theorem 19 intuitively says that the syntactic solution of a pre-equation is maximal among all solutions. Note that, in contrast with equations, Theorem 19 and the theory of pre-equations also work for infinitary trace inclusion.

4.3 Rule formats

A way to instantiate the results in Sections 4.1 and 4.2 is to consider rule formats [1, 16]. These provide a specification for the form of the SOS rules used to describe the constructs of a language. To fit a rule format into the abstract formulation of the theory from Section 4.1, we view the constructs of a language as functions on the states of the LTS (the processes of the language). One of the most common formats is GSOS [3, 29, 9]. For lack of space we only show the instantiation to GSOS of Theorem 8. In the statement, we consider an extension of a GSOS language with constants, in the same way as they appear in CCS, so that the definitions of ‘syntactic solution of equations’ and of ‘innocuous divergence’ can be taken to be the same as for CCS (these are easier to grasp than their formulation in Section 4.1).

Theorem 20. Consider a language whose constructs have SOS rules in the GSOS format and preserve \( \approx \), and an equation \( X = E \) for the language. If \( E \) is autonomous (over the set of functions corresponding to the contexts of the language), and if, in the language extended with constants, the syntactic solution of the equation only has innocuous divergences, then either the equation has no solution or it has a unique solution for \( \approx \).

We briefly discuss the hypotheses in the theorem. Some GSOS rule formats guarantee congruence for weak bisimilarity [29, 9], which allows one to remove the corresponding condition (the condition could actually we weakened, by considering the syntactic positions in which the variables can occur in the equations). Checking the autonomy property is often straightforward; for instance, it holds if, in the body of an equation, all variables are underneath an axiom construct, that is, a construct that (like prefix in CCS) is defined by means of SOS rules in which the set of premises is empty. In the tyft/tyxt formats [11], lookaheads are possible. Lookahead allows one to write rules that ‘look into the future’ (a transition is allowed if certain sequences of actions are possible); this breaks autonomy (condition 3 of Definition 13), hence Theorem 17 is not applicable.
5 Name Passing: the $\pi$-calculus

5.1 Unique solution in the asynchronous $\pi$-calculus

In this section, we port our results onto the asynchronous $\pi$-calculus, $A\pi$ (addressing the full $\pi$-calculus would also be possible, but somewhat more involved). To allow constants (recursive definitions) and equations, we enrich the syntax of $A\pi$ [26] with parametrised processes $(\bar{a})P$. These and the constants form the set of abstractions, ranged over by $F,G$. We omit the definitions of free and bound names. An expression is closed if it does not have free names. Constant definitions are of the form $K \triangleq F$, where $F$ is a closed abstraction. The grammar for processes includes also the application construct $F\langle \bar{a} \rangle$, used to instantiate the formal parameters of the abstraction $F$ with the actual parameters $\bar{a}$.

The body of an equation is also a closed abstraction, possibly containing equation variables. Since the calculus is polyadic, we rely on a sorting system [15] to avoid disagreements in the arities of the tuples of names carried by a given name, and in the parameters of abstractions and equations. For lack of space, we omit the full grammar and the operational semantics (see [24]). When writing examples, for readability, we assume that the syntax contains the guarded replication operator $!a(\bar{b}).P$ (it could be encoded, using constants).

In bisimulations or similar coinductive relations for the asynchronous $\pi$-calculus, no name instantiation is required in the input clause or elsewhere (provided $\alpha$-convertible processes are identified); i.e., the ground versions of the relations are congruence relations [26]. Similarly, the extension of bisimilarity to abstractions only considers fresh names: $F \approx F'$ if $F(\bar{a}) \approx F'(\bar{a})$ where $\bar{a}$ is a tuple of fresh names (as usual, of the appropriate sort).

Theorems 6 and 8 for CCS can be adapted to the asynchronous $\pi$-calculus. The definitions concerning transitions and divergences are transported to $A\pi$ as expected. In the case of an abstraction, one first has to instantiate the parameters with fresh names; thus $F$ has a divergence if the process $F(\bar{a})$ has a divergence, where $\bar{a}$ are fresh names.

► Theorem 21 (Unique solution in $A\pi$). A guarded system of equations whose syntactic solutions do not contain divergences has a unique solution for $\approx$.

► Theorem 22 (Unique solution with innocuous divergences in $A\pi$). A guarded system of equations whose syntactic solutions only have innocuous divergences has a unique solution for $\approx$.

We pointed out in earlier sections on CCS the connection between techniques based on unique solution of equations and ‘up to context’ enhancements of the bisimulation proof method. The same connection is less immediate in name-passing calculi, where indeed there are noticeable differences. In particular, ‘up to context’ enhancements for the ground bisimilarity of the $\pi$-calculus require closure under name instantiation, even when ground bisimilarity is known to be preserved by substitutions (it is an open problem whether the closure can be lifted). Thus, when comparing two derivatives $C[P]$ and $C[Q]$, in general it is not sufficient that $P$ and $Q$ alone are in the candidate relation: one is required to include also all their closures under name substitutions (or, if the terms in the holes are abstractions, instantiation of their parameters with arbitrary tuples of names). In contrast, the two unique solution theorems above are ‘purely ground’: $F = (\bar{z})P$ is solution of an equation $X = (\bar{z})E$ if $P$ and $E\{F/X\}$ are ground bisimilar – a single ground instance of the equation is evaluated.
5.2 An application: encoding of the call-by-name $\lambda$-calculus

To show an extended application of our techniques for the $\pi$-calculus, we revisit the proof of full abstraction for Milner’s encoding of the call-by-name (or lazy) $\lambda$-calculus into $A\pi$ [14] with respect to Lévy Longo Trees (LTs), precisely the completeness part. We use $M, N$ to range over the set $\Lambda$ of $\lambda$-terms, and $x, y, z$ to range over $\lambda$ variables. If $M$ is an open $\lambda$-term, then either $M$ diverges, or $M \Rightarrow \lambda x. M'$, or $M \Rightarrow x M_1 \ldots M_n$.

Definition 23 (Lévy-Longo Tree). The Lévy-Longo Tree (LT) of an open $\lambda$-term $M$, written $LT(M)$, is the (possibly infinite) tree defined coinductively as follows.

1. If $M$ diverges, then $LT(M)$ is the tree with a single node labelled $\bot$.
2. If $M \Rightarrow \lambda x. M'$, then $LT(M)$ is the tree with a root labelled with "$\lambda x.$", and $LT(M')$ as its unique descendant.
3. If $M \Rightarrow x M_1 \ldots M_n$, then $LT(M)$ is the tree with a root labelled with "$x$", and $LT(M_1), \ldots, LT(M_n)$ (in this order) as its $n$ descendants.

LT equality (two $\lambda$-terms are identified if their LTs are equal) can also be presented as a bisimilarity (open bisimilarity, $\simeq^o$), defined as the largest open bisimulation.

Definition 24. A relation $R$ on $\Lambda$ is an open bisimulation if, whenever $M \mathrel{R} N$:

1. $M \Rightarrow \lambda x. M'$ implies $N \Rightarrow \lambda x. N'$ with $M' \mathrel{R} N'$;
2. $M \Rightarrow x M_1 \ldots M_n$ with $n \geq 0$ implies $N \Rightarrow x N_1 \ldots N_n$ and $M_i \mathrel{R} N_i$ for all $1 \leq i \leq n$.
3. The converse of clauses 1 and 2 on the challenges from $N$.

Milner’s encoding is defined thus:

\[
\begin{align*}
\llbracket \lambda x. M \rrbracket & \triangleq (p) p(x, q). \llbracket M \rrbracket(q) \quad \llbracket x \rrbracket \triangleq (p) \nu \pi(p) \\
\llbracket M N \rrbracket & \triangleq (p) \nu r. x \llbracket M \rrbracket(r) \mid \pi(x, p) \mid \llbracket M \rrbracket(q) \llbracket N \rrbracket(q)
\end{align*}
\]

The full abstraction theorem for the encoding [24, 27] states that two $\lambda$-terms have the same LT iff their encodings into $A\pi$ are weakly bisimilar terms. Full abstraction has two components: soundness, which says that if the encodings are weakly bisimilar then the original terms have the same LT; and completeness, which is the converse direction. The proof [24] first establishes some operational correspondence between the behaviour (visible and silent actions) of $\lambda$-terms and of their encodings. Then, exploiting this correspondence, soundness and completeness are proved using the bisimulation proof method. For soundness, this is just open bisimulation (Definition 24). In contrast, completeness involves enhancements of the proof method, notably ‘bisimulation up to context and expansion’. As a consequence, the technique requires having developed the basic theory for the expansion preorder (e.g., congruence properties and basic algebraic laws), and requires an operational correspondence fine enough in order to be able to reason about expansion.

Below we show that, by appealing to unique solution of equations, completeness can be proved by defining an appropriate system of equations, each of which having a simple shape, and without the need for auxiliary preorders. For this, the only results needed are: (i) validity of $\beta$-reduction for the encoding (Lemma 25), whose proof is simple and consists in the application of a few algebraic laws (including laws for replication); (ii) the property that if $M$ diverges then $\llbracket M \rrbracket(p)$ may never produce a visible action [24].

Lemma 25 (Validity of $\beta$-reduction, [24]). For $M \in \Lambda$, if $M \Rightarrow M'$ then $\llbracket M \rrbracket \approx \llbracket M' \rrbracket$.

Theorem 26 (Completeness, [24]). For $M, N \in \Lambda$, $LT(M) = LT(N)$ implies $\llbracket M \rrbracket \approx \llbracket N \rrbracket$.

Proof. Suppose $V$ and $W$ are two $\lambda$-terms with the same LT. We define a system of equations, solutions of which are obtained from the encodings of $V$ and $W$. We will then use Theorem 22 to deduce $\llbracket V \rrbracket \approx \llbracket W \rrbracket$. If $V$ and $W$ have the same LT, then there is an open bisimulation $R$.
containing the pair \((V, W)\). The variables of the equations are of the form \(X_{M,N}\) for \(M \not\equiv N\), and there is one equation for each pair in \(\mathcal{R}\). We show how the equation for a pair \(M \not\equiv N\) is built. We assume an ordering of the \(\lambda\)-calculus variables so to be able to view a finite set of variables as a tuple. Thus we use \(\tilde{x}\) for the variables appearing free in \(M\) or \(N\).

Essentially, the equations are the translation of the clauses of Definition 24, assuming a generalisation of the encoding to equation variables:

- If \(M, N\) are both divergent, then the equation is \(X_{M,N} = (\tilde{x}, p)!\tau\).
- If \(M, N\) satisfy clause 1 of Definition 24, the equation is \(X_{M,N} = (\tilde{x}, p)p(x, q).X_{M',N'}(\tilde{y}, q)\), where \(\tilde{y}\) are the free variables in \(M', N'\).
- Finally, if \(M, N\) satisfy clause 2 of Definition 24, the equation is given by the translation of \(x \ X_{M_1,N_1} \ldots X_{M_n,N_n}\), which, rearranging restrictions and parallel compositions, is

\[
X_{M,N} = (\tilde{x}, p)(\nu r_0, \ldots, r_n)(\pi r_0 | \pi_1 r_1, x_1 | \ldots | \pi_{n-1} r_n, x_n | \lambda x_1(q_1).X_{M_1,N_1}(\tilde{x}_1, q_1) | \ldots | \lambda x_n(q_n).X_{M_n,N_n}(\tilde{x}_n, q_n))
\]

where \(\tilde{x}_i\) are the free variables in \(M_i, N_i\). Applying Lemma 9 (more precisely, its \(A\pi\) analogue, with additional reference to the sorting system to handle name instantiation in abstractions and inputs), we show that the equations may only produce innocuous divergences.

Now, for \((M, N) \in \mathcal{R}\), set \(F_{M,N}\) to be the abstraction \((\tilde{x}, p)[M](p)\), and similarly \(G_{M,N} \equiv (\tilde{x}, p)[N](p)\). The set of all such abstractions \(F_{M,N}\) yields a solution for the system of equations, and the same for the \(G_{M,N}\)’s. The proof that they are solutions (hence bisimilar) is a consequence of Lemma 25. For instance, for clause 1 of Definition 24, we have:

\[
F_{M,N} = (\tilde{x}, p)[M](p) \approx (\tilde{x}, p)[\lambda x. M'](p) \approx (\tilde{x}, p)[\lambda x. X_{M',N'}](p)\{F_{M',N'}/X_{M',N'}\}
\]

### 6 Future Work

We have compared our techniques to one of the most powerful forms of enhancements of the bisimulation proof method, namely Pous ‘up to transitivity and context’, showing that, up to a technical condition, our techniques are at least as powerful. We believe that also the converse holds, though possibly under different side conditions. We leave a detailed analysis of this comparison, which seems non-trivial, for future work. In this respect, the goal of the work on unique solution of equations is to provide a way of better understanding up-to techniques and to shed light into the conditions for their soundness. Pous technique, for instance, is arguably more complex, both in its definition and its application.

Another aspect in which a deep comparison with up-to techniques might be useful is understanding the need for the closure under substitutions in up to context techniques for name-passing calculi such as the asynchronous \(\pi\)-calculus. Such a closure is rather heavy and is a long-standing open problem. However, currently it is unclear how to formally relate bisimulation enhancements and ‘unique solution of equations’ in name-passing calculi.

Up-to techniques have been analysed in an abstract setting using lattice theory [18] and category theory [4, 23]. It could be interesting to do the same for the unique-solution techniques, to study their connections with up-to techniques, and which equivalences can be handled (possibly using, or refining, the abstract formulation of Section 4).

In comparison with the enhancements of the bisimulation proof method, the main drawback of the techniques exposed in this paper is the presence of a semantic condition, involving divergence: the unfoldings of the equations should not produce divergences, or
only produce innocuous divergences. A syntactic condition for this has been proposed (Lemma 9). Various techniques for checking divergence exist in the literature, including type-based techniques [30, 7]. However, in general divergence is undecidable, and therefore, the check may sometimes be unfeasible. Nevertheless, the equations that one writes for proofs usually involve forms of ‘normalised’ processes, and as such they are divergence free (or at most, contain only innocuous divergences). More experiments are needed to validate this claim or to understand how limiting this problem is.

References

Divergence and Unique Solution of Equations


Controlling a Population

Nathalie Bertrand\textsuperscript{1}, Miheer Dewaskar\textsuperscript{2}, Blaise Genest\textsuperscript{3}, and Hugo Gimbert\textsuperscript{4}

\begin{enumerate}
\item Inria, IRISA, Rennes, France
\item University of North Carolina, Chapel Hill, USA
\item CNRS, IRISA, Rennes, France
\item CNRS, LaBRI, Bordeaux, France
\end{enumerate}

\textbf{Abstract}

We introduce a new setting where a population of agents, each modelled by a finite-state system, are controlled uniformly: the controller applies the same action to every agent. The framework is largely inspired by the control of a biological system, namely a population of yeasts, where the controller may only change the environment common to all cells. We study a synchronisation problem for such populations: no matter how individual agents react to the actions of the controller, the controller aims at driving all agents synchronously to a target state. The agents are naturally represented by a non-deterministic finite state automaton (NFA), the same for every agent, and the whole system is encoded as a 2-player game. The first player (Controller) chooses actions, and the second player (Agents) resolves non-determinism for each agent. The game with $m$ agents is called the $m$-population game. This gives rise to a parameterized control problem (where control refers to 2 player games), namely the population control problem: can Controller control the $m$-population game for all $m \in \mathbb{N}$ whatever Agents does?

In this paper, we prove that the population control problem is decidable, and it is an EXPTIME-complete problem. As far as we know, this is one of the first results on parameterized control. Our algorithm, not based on cut-off techniques, produces winning strategies which are symbolic, that is, they do not need to count precisely how the population is spread between states. We also show that if there is no winning strategy, then there is a population size $M$ such that Controller wins the $m$-population game if and only if $m \leq M$. Surprisingly, $M$ can be doubly exponential in the number of states of the NFA, with tight upper and lower bounds.

\textbf{1998 ACM Subject Classification} D.2.4 Software/Program Verification

\textbf{Keywords and phrases} Model-checking, control, parametric systems

\textbf{Digital Object Identifier} 10.4230/LIPIcs.CONCUR.2017.12

\section{Introduction}

Finite-state controllers, implemented by software, find applications in many different domains: telecommunication, planes, etc. There have been many theoretical studies from the model checking community to show that finite-state controllers are sufficient to control systems in idealised settings. Usually, the problem would be modeled as a game: some players model the controller, and some players model the system [5], the game settings (number of players, their power, their observation) depending on the context.

Lately, finite-state controllers have been used to control living organisms, such as a population of yeasts [23]. In this application, microscopy is used to monitor the fluorescence

\textsuperscript{*} This work was partially supported by ANR project STOCH-MC (ANR-13-BS02-0011-01).
level of a population of yeasts, reflecting the concentration of some molecule, which differs from cell to cell. Finite-state systems can model a discretisation of the population of yeasts [23, 3]. A controller decides the frequency and duration of injections of a sorbitol solution, uniform over the yeast population. However, the response of each cell to the osmotic stress induced by sorbitol varies, influencing the concentration of the fluorescent molecule. The objective is to control the population to drive it through a sequence of predetermined fluorescence states.

In this paper, we model this system of yeasts in an idealised setting: we require the (perfectly-informed) controller to surely lead synchronously all agents of a population to a state (one of the predetermined fluorescence states). Such a population control problem does not fit in traditional frameworks from the model checking community. We thus introduce the $m$-population game, where a population of $m$ identical agents is controlled uniformly. Each agent is modeled as a nondeterministic finite-state automaton (NFA), the same for each agent. The first player, called Controller, applies the same action, a letter from the NFA alphabet, to every agent. Its opponent, called Agents, chooses the reaction of each individual agent. These reactions can be different due to non determinism. The objective for Controller is to gather all agents synchronously in the target state (which can be a sink state w.l.o.g.), and Agents seeks the opposite objective. While this idealised setting may not be entirely satisfactory, it constitutes a simple setting, as a first step towards more complex settings.

Dealing with large populations explicitly is in general intractable due to the state-space explosion problem. We thus consider the associated symbolic parameterized control problem, asking to reach the goal independently of the population size. We prove that this problem is decidable. While parameterized verification received recently quite some attention (see related work), our results are one of the first on parameterized control, as far as we know.

Our results. We first show that considering an infinite population is not equivalent to the parameterized control problem for all non zero integer $m$: there are cases where Controller cannot control an infinite population but can control every finite population. Solving the $\infty$-population game reduces to checking a reachability property on the support graph [21], which can be easily done in $\text{PSPACE}$. On the other hand, solving the parameterized control problem requires new proof techniques, data structures and algorithms.

We easily obtain that when the answer to the population control problem is negative, there exists a population size $M$, called the cut-off, such that Controller wins the $m$-population game if and only if $m \leq M$. Surprisingly, we obtain a lower-bound on the cut-off doubly exponential in the number of states of the NFA. Following usual cut-off techniques would thus yield an inefficient algorithm of complexity at least $2^{\text{EXPTIME}}$.

To obtain better complexity, we developed new proof techniques (not based on cut-off techniques). Using them, we prove that the population control problem is $\text{EXPTIME}$-complete. As a byproduct, we obtain a doubly exponential upper-bound for the cut-off, matching the lower-bound. Our techniques are based on a reduction to a parity game with exponentially many states and a polynomial number of priorities. The parity game gives some insight on the winning strategies of Controller in the $m$-population games. Controller selects actions based on a set of transfer graphs, giving for each current state the set of states at time $i$ from which agent came from, for different values of $i$. We show that it suffices for Controller to remember at most a quadratic number of such transfer graphs, corresponding to a quadratic number of indices $i$. If Controller wins this parity game then he can uniformly apply his winning strategy to all $m$-population games, just keeping track of these transfer graphs, independently of the exact count in each state. If Agents wins the parity game then he also has a uniform winning strategy in $m$-population games, for $m$ large enough, which consists
in splitting the agents evenly among all transitions of the transfer graphs. Missing proofs are available in the research report [6].

**Related work.** Parameterized verification of systems with many identical components started with the seminal work of German and Sistla in the early nineties [16], and received recently quite some attention. The decidability and complexity of these problems typically depend on the communication means, and on whether the system contains a leader (following a different template) as exposed in the recent survey [13]. This framework has been extended to timed automata templates [2, 1] and probabilistic systems with Markov decision processes templates [7, 8]. Another line of work considers population protocols [4, 15]. Close in spirit are broadcast protocols [14], in which one action may move an arbitrary number of agents from one state to another. Our model can be modeled as a subclass of broadcast protocols, where broadcasts emissions are self loops at a unique state, and no other synchronisation allowed. The parameterized reachability question considered for broadcast protocols is trivial in our framework, while our parameterized control question would be undecidable for broadcast protocols. In these different works, components interact directly, while in our work, the interaction is indirect via the common action of the controller. Further, the problems considered in related work are pure verification questions, and do not tackle the difficult issue of synthesising a controller for all instances of a parameterized system, which we do.

There are very few contributions pertaining to parameterized games with more than one player. The most related is [20], which proves decidability of control of mutual exclusion-like protocols in the presence of an unbounded number of agents. Another contribution in that domain is the one of broadcast networks of identical parity games [8]. However, the game is used to solve a verification (reachability) question rather than a parametrized control problem as in our case. Also the roles of the two players are quite different.

The winning condition we are considering is close to *synchronising words*. The original synchronising word problem asks for the existence of a word \( w \) and a state \( q \) of a deterministic finite state automaton, such that no matter the initial state \( s \), reading \( w \) from \( s \) would lead to state \( q \) (see [24] for a survey). Lately, synchronising words have been extended to NFAs [21]. Compared to our settings, the author assumes a possibly infinite population of agents, who could leak arbitrarily often from a state to another. The setting is thus not parametrized, and a usual support arena suffices to obtain a \textit{PSPACE} algorithm. Synchronisation for probabilistic models [11, 12] have also been considered: the population of agents is not finite nor discrete, but rather continuous, represented as a distribution. The distribution evolves deterministically with the choice of the controller (the probability mass is split according to the probabilities of the transitions), while in our setting, each agent can non deterministically move. In [11], the controller needs to apply the same action whatever the state the agents are in (like our setting), and then the existence of a controller is undecidable. In [12], the controller can choose the action depending on the state each agent is in (unlike our setting), and the existence of a controller reaching uniformly a set of states is \textit{PSPACE}-complete.

Last, our parameterized control problem can be encoded as a 2-player game on VASS [9], with one counter per state of the NFA: the opponent gets to choose the population size (a counter value), and the move of each agent corresponds to decrementing a counter and incrementing another. Such a reduction yields a symmetrical game on VASS in which both players are allowed to modify the counter values, in order to check that the other player did not cheat. Symmetrical games on VASS are undecidable [9], and their asymmetric variant (in which only one player is allowed to change the counter values) are decidable in \textit{2EXPTIME} [19], thus with higher complexity than our specific parameterized control problem.
2 The population control problem

2.1 The \( m \)-population game

A nondeterministic finite automaton (NFA for short) is a tuple \( \mathcal{A} = (Q, \Sigma, q_0, \Delta) \) with \( Q \) a finite set of states, \( \Sigma \) a finite alphabet, \( q_0 \in Q \) an initial state, and \( \Delta \subseteq Q \times \Sigma \times Q \) the transition relation. We assume throughout the paper that NFAs are complete, that is, \( \forall q \in Q, a \in \Sigma, \exists p \in Q : (q, a, p) \in \Delta \). In the following, incomplete NFAs, especially in figures, have to be understood as completed with a sink state.

For every integer \( m \), we consider a system \( \mathcal{A}^m \) with \( m \) identical agents \( A_1, \ldots, A_m \) of the NFA \( \mathcal{A} \). The system \( \mathcal{A}^m \) is itself an NFA \( (Q^m, \Sigma, q_0^m, \Delta^m) \) defined as follows. Formally, states of \( \mathcal{A}^m \) are called configurations, and they are tuples \( q = (q_1, \ldots, q_m) \in Q^m \) describing the current state of each agent in the population. We use the shorthand \( q_i[m] \), or simply \( q_i \) when \( m \) is clear from context, to denote the initial configuration \((q_0, \ldots, q_0)\) of \( \mathcal{A}^m \). Given a target state \( f \in Q \), the \( f \)-synchronizing configuration is \( f^m = (f, \ldots, f) \) in which each agent is in the target state.

The intuitive semantics of \( \mathcal{A}^m \) is that at each step, the same action from \( \Sigma \) applies to all agents. The effect of the action however may not be uniform given the nondeterminism present in \( \mathcal{A} \): we have \((q_1, \ldots, q_m), a, (q'_1, \ldots, q'_m)) \in \Delta^m \) if \((q_j, a, q'_j) \in \Delta \) for all \( j \leq m \). A (finite or infinite) play in \( \mathcal{A}^m \) is an alternating sequence of configurations and actions, starting in the initial configuration: \( \pi = (q_0^0 a_0 q_1^0 a_1 \cdots) \) such that \( (q_i, a_i, q_{i+1}) \in \Delta^m \) for all \( i \).

This is the \( m \)-population game between Controller and Agents, where Controller chooses the actions and Agents chooses how to resolve non-determinism. The objective for Controller is to gather all agents synchronously in \( f \) while Agents seeks the opposite objective.

Our parameterized control problem asks whether Controller can win the \( m \)-population game for every \( m \in \mathbb{N} \). A strategy of Controller in the \( m \)-population game is a function mapping finite plays to actions, \( \sigma : (Q^m \times \Sigma)^* \times Q^m \rightarrow \Sigma \). A play \( \pi = q_0^0 a_0 q_1^0 a_1 q_2^0 \cdots \) is said to respect \( \sigma \), or is a play under \( \sigma \), if it satisfies \( a_i = \sigma(q_0^0 a_0 q_1^0 a_1 q_2^0 \cdots) \) for all \( i \in \mathbb{N} \). A play \( \pi = q_0^0 a_0 q_1^0 a_1 q_2^0 \cdots \) is winning if it hits the \( f \)-synchronizing configuration, that is \( q_i = f^m \) for some \( j \in \mathbb{N} \). Controller wins the \( m \)-population game if he has a strategy such that all plays under this strategy are winning. One can assume without loss of generality that \( f \) is a sink state. If not, it suffices to add a new action leading tokens from \( f \) to the new target sink state \( \odot \) and tokens from other states to a losing sink state \( \odot \). The goal of this paper is to study the following parameterized control problem:

<table>
<thead>
<tr>
<th>Population control problem</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> An NFA ( \mathcal{A} = (Q, q_0, \Sigma, \Delta) ) and a target state ( f \in Q ).</td>
</tr>
<tr>
<td><strong>Output:</strong> Yes iff for every integer ( m ) Controller wins the ( m )-population game.</td>
</tr>
</tbody>
</table>

For a fixed \( m \), the winner of the \( m \)-population game can be determined by solving the underlying reachability game with \( |Q|^m \) states, which is intractable for large values of \( m \). On the other hand, the answer to the population control problem gives the winner of the \( m \)-population game for arbitrary large values of \( m \). To obtain a decision procedure for this parameterised problem, new data structures and algorithmic tools need to be developed, much more elaborate than the standard algorithm solving reachability games.

**Example 1.** We illustrate the population control problem with the example \( \mathcal{A}_{\text{split}} \) on alphabet \( \{a, b, \delta\} \) in Figure 1. Here, to represent configurations we use a counting abstraction, and identify \( q \) with the vector \((n_0, n_1, n_2, n_3)\), where \( n_0 \) is the number of agents in state \( q_0 \), and so on. Under these notations, there is a way to gather agents synchronously to \( f \). We can
give a symbolic representation of a memoryless winning strategy \( \sigma \): \( \forall k_0, k_1 > 0, \forall k_2, k_3 \geq 0, \sigma(k_0, 0, 0, k_3) = \delta, \sigma(0, k_1, k_2, k_3) = a, \sigma(0, 0, k_2, k_3) = b. \) Indeed, the number of agents outside \( f \) decreases by at least one at every other step. The properties of this example will be detailed later and play a part in proving a lower bound (see Proposition 20).

2.2 Parameterized control and cut-off

A first observation for the population control problem is that \( q_0 [m], f^{m} \) and \( Q^{m} \) are stable under a permutation of coordinates. A consequence is that the \( m \)-population game is also symmetric under permutation, and thus the set of winning configurations is symmetric and the winning strategy can be chosen uniformly from symmetric winning configurations. Therefore, if Controller wins the \( m \)-population game then he has a positional winning strategy which only counts the number of agents in each state of \( A \) (the counting abstraction used in Example 1).

\[ \text{Proposition 2.} \text{ Let } m \in \mathbb{N}. \text{ If Controller wins the } m \text{-population game, then he wins the } m' \text{-population game for every } m' \leq m. \]

The idea to define \( \sigma_{m'} \) is to simulate the missing \( m-m' \) agents arbitrarily and apply \( \sigma_m \).

Hence, when the answer to the population control problem is negative, there exists a cut-off, that is a value \( M \in \mathbb{N} \) such that for every \( m < M \), Controller has a winning strategy in \( A^m \), and for every \( m \geq M \), he has no winning strategy.

\[ \text{Example 3.} \text{ To illustrate the notion of cut-off, consider the NFA on alphabet } A \cup \{ b \} \text{ from Figure 2. Unspecified transitions lead to a sink state } \odot. \]

The cut-off is \( M = |Q| - 2 \) in this case. Indeed, we have the following two directions:

On the one hand, for \( m < M \), there is a winning strategy \( \sigma_m \) in \( A^m \) to reach \( f^m \), in just two steps. It first plays \( b \), and because \( m < M \), in the next configuration, there is at least one state \( q_i \) such that no agent is in \( q_i \). It then suffices to play \( a_i \) to win.

Now, if \( m \geq M \), there is no winning strategy to synchronize in \( f \), since after the first \( b \), agents can be spread so that there is at least one agent in each state \( q_i \). From there, Controller can either play action \( b \) and restart the whole game, or play any action \( a_i \), leading at least one agent to the sink state \( \odot \).

2.3 Main results

Our main result is the decidability of the population control problem:
Theorem 4. The population control problem is EXPTIME-complete.

When the answer to the population control problem is positive, there exists a symbolic strategy \( \sigma \), applicable to all instances \( A^m \), that does not need to count the number of agents in each state. This symbolic strategy requires exponential memory. Otherwise, the cut-off is at most doubly exponential, which is asymptotically tight.

Theorem 5. In case the answer to the population control problem is negative, the cut-off is at most \( \leq 2^{2^{|Q|^{4}}} \). There is a family of NFA \( (A_n) \) of size \( O(n) \) and whose cut-off is \( 2^n \).

3 The capacity game

The objective of this section is to show that the population control problem is equivalent to solving a game called the capacity game. To introduce useful notations, we first recall the population game with infinitely many agents, as studied in [21] (see also [22] p.81).

3.1 The \( \infty \)-population game

To study the \( \infty \)-population game, the behaviour of infinitely many agents is abstracted into supports which keep track of the set of states in which at least one agent is. We thus introduce the support game, which relies on the notion of transfer graphs. Formally, a transfer graph is a subset of \( Q \times Q \) describing how agents are moved during one step. The domain of a transfer graph \( G \) is \( \text{Dom}(G) = \{ q \in Q \mid \exists (q, r) \in G \} \) and its image is \( \text{Im}(G) = \{ r \in Q \mid \exists (q, r) \in G \} \). Given an NFA \( A = (Q, \Sigma, q_0, \Delta) \) and \( a \in \Sigma \), the transfer graph \( G \) is compatible with \( a \) if for every edge \( (q, r) \) of \( G \), \( (q, a, r) \in \Delta \). We write \( \mathcal{G} \) for the set of transfer graphs.

The support game of an NFA \( A \) is a two-player reachability game played by Controller and Agents on the support arena as follows. States are supports, i.e., non-empty subsets of \( Q \) and the play starts in \( \{q_0\} \). The goal support is \( \{f\} \). From a support \( S \), first Controller chooses a letter \( a \in \Sigma \), then Agents chooses a transfer graph \( G \) compatible with \( a \) and such that \( \text{Dom}(G) = S \), and the next support is \( \text{Im}(G) \). A play in the support arena is described by the sequence \( \rho = S_0 \xrightarrow{a_1, G_1} S_1 \xrightarrow{a_2, G_2} \ldots \) of supports and actions (letters and transfer graphs) of the players. Here, Agents best strategy is to play the maximal graph possible (this is not the case with discrete populations), and we obtain a PSPACE-complete algorithm [21]:

Proposition 6. Controller wins the \( \infty \)-population game iff he wins the support game.

This result cannot be used for deciding the population control problem, because Controller might win every \( m \)-population game (with \( m < \infty \)) and at the same time lose the \( \infty \)-population game. For that, consider the example from Figure 1. As already shown, Controller wins any \( m \)-population game with \( m < \infty \). However, Agents can win the \( \infty \)-population game by splitting agents from \( q_0 \) to both \( q_1 \) and \( q_2 \) each time Controller plays \( \delta \). This way, the sequence of supports is \( \{q_0\} \{q_1, q_2\} \{\{q_0, f\}\{q_1, q_2, f\}\}^* \), which never hits \( \{f\} \).
3.2 Realisable plays

Plays of the $m$-population game (for $m < \infty$) can be abstracted as plays in the support game, by forgetting the identity of agents and keeping only track of edges that are used by at least one agent. Formally, given a play $\pi = q_0a_0q_1a_1q_2\cdots$ of the $m$-population game, define for every integer $n$, $S_n = \{q_n[i] \mid 1 \leq i \leq m\}$ and $G_{n+1} = \{(q_n[i], q_{n+1}[i]) \mid 1 \leq i \leq m\}$. We denote $\Phi_m(\pi)$ the play $S_0 \xrightarrow{a_1,G_1} S_1 \xrightarrow{a_2,G_2} \cdots$ in the support arena, called the projection of $\pi$.

Not every play in the support arena can be obtained by projection. This is the reason for introducing the notion of realisable plays:

Definition 7 (Realisable plays). A play of the support game is realisable if there exists $m < \infty$ such that it is the projection by $\Phi_m$ of a play in the $m$-population game.

To characterise realisability, we introduce entries of accumulators:

Definition 8. Let $\rho = S_0 \xrightarrow{a_1,G_1} S_1 \xrightarrow{a_2,G_2} \cdots$ be a play in the support arena. An accumulator of $\rho$ is a sequence $T = (T_j)_{j \in \mathbb{N}}$ such that for every integer $j$, $T_j \subseteq S_j$, and which is successor-closed i.e., for every $j \in \mathbb{N}$, $(s \in T_j \land (s, t) \in G_{j+1}) \implies t \in T_{j+1}$. For every $j \in \mathbb{N}$, an edge $(s, t) \in G_{j+1}$ is an entry to $T$ if $s \not\in T_j$ and $t \in T_{j+1}$.

Definition 9 (Plays with finite and bounded capacity). A play has finite capacity if all its accumulators have finitely many entries, infinite capacity otherwise, and bounded capacity if the number of entries of its accumulators is bounded.

Realisability is actually equivalent to bounded capacity:

Lemma 10. A play is realisable iff it has bounded capacity.

An example is given on Figure 3 which represents an NFA, two transfer graphs $G$ and $H$, and a play $GHG^2HG^3\cdots$. Obviously, this play is not realisable because at least $n$ agents are needed to realise $n$ transfer graphs $G$ in a row: at each $G$ step, at least one agent moves from $q_0$ to $q_1$, and no new agent enters $q_0$. A simple analysis shows that there are only two kinds of non-trivial accumulators $(T_j)_{j \in \mathbb{N}}$ depending on whether their first non-empty $T_j$ is $\{q_0\}$ or $\{q_1\}$. We call these top and bottom accumulators, respectively. All accumulators have finitely many entries, thus the play has finite capacity. However, for every $n \in \mathbb{N}$ there is a bottom accumulator with $2n$ entries. As an example, a bottom accumulator with 4 entries (in red) is depicted on the figure. Therefore, the capacity of this play is not bounded.

3.3 The capacity game

An idea to obtain a game on the support arena equivalent with the population control problem is to make Agents lose whenever the play is not realisable, i.e. whenever the play...
has unbounded capacity. One issue with (un)bounded capacity is however that it is not a regular property for runs. Hence, it is not easy to use it as a winning condition. On the contrary, finite capacity is a regular property. We thus relax (un)bounded capacity using (in)finite capacity and define the corresponding abstraction of the population game:

**Definition 11 (Capacity game).** The capacity game is the game played on the support arena, where Controller wins a play if either the play reaches \( \{ f \} \) or the play has infinite capacity. A player wins the capacity game if he has a winning strategy in this game.

We show that this relaxation can be used to decide the population control problem.

**Theorem 12.** The answer to the population control problem is positive iff Controller wins the capacity game.

This theorem is a direct corollary of the following proposition:

**Proposition 13.** Either Controller or Agents wins the capacity game, and the winner has a winning strategy with finite memory. In case Controller is the winner of the capacity game, he wins all \( \{ m \} \)-population games, for every integer \( m \). In case Agents wins the capacity game with a strategy with finite memory of size \( M \), he wins the \( |Q|^{1+|M|/4^{|Q|}} \)-population game.

**Proof of first and second assertions.** We start with the first assertion. Whether a play has infinite capacity can be verified by a non-deterministic Büchi automaton of size \( 2^{|Q|} \) on the alphabet of transfer graphs, which guesses an accumulator on the fly and checks that it has infinitely many entries. This Büchi automaton can be determined into a parity automaton (e.g. using Safra’s construction) with state space \( M \) of size \( O(2^{|Q|}) \). The synchronized product of this deterministic parity automaton with the support game produces a parity game which is equivalent with the capacity game, in the sense that, up to unambiguous synchronization with the deterministic automaton, plays and strategies in both games are the same and the synchronization preserves winning plays and strategies. Since parity games are determined and positional [25], either Controller or Agents has a positional winning strategy in the parity game, thus either Controller or Agents has a winning strategy with finite memory \( M \) in the capacity game.

Let us prove the second assertion. Assuming that Controller wins the capacity game with a strategy \( \sigma \), he can win any \( \{ m \} \)-population game, \( m < \infty \), with the strategy \( \sigma_m = \sigma \circ \Phi_m \). The projection \( \Phi_m(\pi) \) of every infinite play \( \pi \) respecting \( \sigma_m \) is realisable, thus \( \Phi_m(\pi) \) has bounded, hence finite, capacity (Lemma 10). Moreover \( \Phi_m(\pi) \) respects \( \sigma \), and since \( \sigma \) wins the capacity game, \( \Phi_m(\pi) \) reaches \( \{ f \} \). Thus \( \pi \) reaches \( f^m \) and \( \sigma_m \) is winning.

The last assertion is proved in [6]. □

As consequence of Proposition 13, the population control problem can be decided by explicitly computing the parity game and solving it, in \( 2\text{EXPTIME} \). In the next section we will improve this complexity bound to \( \text{EXPTIME} \).

We conclude with an example showing that, in general, positional strategies are not sufficient to win the capacity game. Consider the example of Figure 4, where the only way for Controller to win is to reach a support without \( q_2 \) and play \( c \). With a memoryless strategy, Controller cannot win the capacity game. There are only two memoryless strategies from support \( S = \{ q_1, q_2, q_3, q_4 \} \). If Controller only plays \( a \) from \( S \), the support remains \( S \) and the play has bounded capacity. If he only plays \( b \)'s from \( S \), then Agents can split tokens from \( q_3 \) to both \( q_2, q_4 \) and the play remains in support \( S \), with bounded capacity. In both cases, the play has finite capacity and Controller loses.
However, Controller can win the capacity game. His (finite-memory) winning strategy $\sigma$ consists in first playing $c$, and then playing alternatively $a$ and $b$, until the support does not contain $\{q_2\}$, in which case he plays $c$ to win. Two consecutive steps $ab$ send $q_2$ to $q_1$, $q_1$ to $q_3$, $q_3$ to $q_1$, and $q_4$ to either $q_4$ or $q_2$. To prevent Controller from playing $c$ and win, Agents needs to spread from $q_4$ to both $q_4$ and $q_2$ every time $ab$ is played. Consider the accumulator $T$ defined by $T_{2i} = \{q_1, q_2, q_3\}$ and $T_{2i-1} = \{q_1, q_2, q_4\}$ for every $i > 0$. It has an infinite number of entries (from $q_4$ to $T_{2i}$). Hence Controller wins if this play is executed. Else, Agents eventually keeps all agents from $q_4$ in $q_4$ when $ab$ is played, implying the next support does not contain $q_2$. Strategy $\sigma$ is thus a winning strategy for Controller.

4 Solving the capacity game in EXPTIME

To solve efficiently the capacity game, we build an equivalent exponential size parity game with a polynomial number of parities. To do so, we enrich the support arena with a tracking list responsible of checking whether the play has finite capacity. The tracking list is a list of transfer graphs, which are used to detect certain patterns called leaks.

4.1 Leaking graphs

In order to detect whether a play $\rho = S_0 \xrightarrow{a_1} G_1 S_1 \xrightarrow{a_2} G_2 \ldots$ has finite capacity, it is enough to detect leaking graphs (characterising entries of accumulators). Further, leaking graphs have special separation properties which will allow us to track a small number of graphs. For $G, H$ two graphs, we denote $(a, b) \in G \cdot H$ iff there exists $z$ with $(a, z) \in G$, and $(z, b) \in H$.

Definition 14 (Leaks and separations). Let $G, H$ be two transfer graphs. We say that $G$ leaks at $H$ if there exist states $q, x, y$ with $(q, y) \in G \cdot H$, $(x, y) \in H$ and $(q, x) \notin G$. We say that $G$ separates a pair of states $(r, t)$ if there exists $q \in Q$ with $(q, r) \in G$ and $(q, t) \notin G$.

The tracking list will be composed of concatenated graphs $i$ of the form $G[i,j] = G_{i+1} \cdots G_j$ relating $S_i$ with $S_j$: $(s_i, s_j) \in G[i,j]$ if there exists $(s_k)_{i<k<j}$ with $(s_k, s_{k+1}) \in$
Controlling a Population

$G_{k+1}$ for all $i \leq k \leq j$. Infinite capacity relates to leaks in the following way:

**Lemma 15.** A play has infinite capacity iff there exists an index $i$ such that $G[i,j]$ leaks at $G_{j+1}$ for infinitely many indices $j$.

In this case, we say that index $i$ leaks infinitely often. Note that if $G$ separates $(r,t)$, and $r,t$ have a common successor by $H$, then $G$ leaks at $H$. To link leaks with separations, we consider for each index $k$, the pairs of states that have a common successor, in possibly several steps, as expressed by the symmetric relation $R_k$: $(r,t) \in R_k$ iff there exists $j \geq k$ and $y \in Q$ such that $(r,y) \in G[k,j] \land (t,y) \in G[k,j]$.

**Lemma 16.** For $i < n$ two indices, the following three properties hold:
1. If $G[i,n]$ separates $(r,t) \in R_n$, then there exists $m \geq n$ such that $G[i,m]$ leaks at $G_{m+1}$.
2. If index $i$ does not leak infinitely often, then the number of indices $j$ such that $G[i,j]$ separates some $(r,t) \in R_j$ is finite.
3. If index $i$ leaks infinitely often, then for all $j > i$, $G[i,j]$ separates some $(r,t) \in R_j$.

**4.2 The tracking list**

The tracking list exploits the relationship between leaks and separations. It is a list of transfer graphs which altogether separate all possible pairs, and are sufficient to detect when leaks occur. Notice that telling at step $j$ whether the pair $(r,t)$ belongs to $R_j$ cannot be performed by a deterministic automaton. We thus a priori have to consider every pair $(r,t) \in Q^2$ for separation. The tracking list $L_n$ at step $n$ is defined inductively as follows. $L_0$ is the empty list, and for $n > 0$, the list $L_n$ is computed in three stages:
1. first, every graph $H$ in the list $L_{n-1}$ is concatenated with $G_n$, yielding $H \cdot G_n$;
2. second, $G_n$ is added at the end of the obtained list;
3. last, the list is filtered: a graph $H$ is kept if and only if it separates a pair of states $(p,q) \in Q^2$ which is not separated by any graph that appears earlier in the list.

Because of the third item, there are at most $|Q|^2$ graphs in the tracking list. The list may become empty if no pair of states is separated by any graph, for example if all the graphs are complete. Let $L_n = \{ H_1, \ldots, H_r \}$ be the tracking list at step $n$. Then each transfer graph $H_r \in L_n$ is of the form $H_r = G[t_r,n]$. We say that $r$ is the level of $H_r$, and $t_r$ the index tracked by $H_r$. Observe that the lower the level of a graph in the list, the smaller the index it tracks. When we consider the sequence of tracking lists $(L_n)_{n \in \mathbb{N}}$, for every index $i$, either it eventually stops to be tracked or it is tracked forever from step $i$, i.e. for every $n \geq i$, $G[i,n]$ belongs to $L_n$. In the latter case, $i$ is said to be remanent (because it will never disappear).

Using Lemma 15 and the second and third statements of Lemma 16, we obtain:

**Lemma 17.** A play has infinite capacity iff there exists an index $i$ such that $i$ is remanent and leaks infinitely often.

**4.3 The parity game**

We now describe a parity game $P\mathcal{G}$, which extends the support arena with on-the-fly computation of the tracking list.

**Priorities.** By convention, lowest priorities are the most important and the odd parity is good for Controller, so Controller wins iff the lim inf of the priorities is odd. With each level $1 \leq r \leq |Q|^2$ of the tracking list are associated two priorities $2r$ and $2r + 1$, and on top of that are added priorities $1$ and $2|Q|^2 + 2$, hence the set of all priorities is $\{1, \ldots, 2|Q|^2 + 2\}$.
When Agents chooses a transition labelled by a transfer graph $G$, the tracking list is updated with $G$ and the priority of the transition is determined as the smallest among: priority 1 if the support $\{f\}$ has ever been visited, priority $2r + 1$ for the smallest $r$ such that $H_r$ (from level $r$) leaks at $G$, priority $2r$ for the smallest level $r$ where a graph was removed, and in all other cases priority $2|Q|^2 + 2$.

**States and transitions.** $G^{\leq |Q|^2}$ denotes the set of list of at most $|Q|^2$ transfer graphs.

- States of $PG$ form a subset of $\{0, 1\} \times 2^Q \times G^{\leq |Q|^2}$, each state being of the form $(b, S, H_1, \ldots, H_\ell)$ with $b \in \{0, 1\}$ a bit indicating whether a support in $\{f\}$ has been seen, $S$ the current support and $(H_1, \ldots, H_\ell)$ the tracking list. The initial state is $(0, (\emptyset), \emptyset)$.

- Transitions in $PG$ are all $(b, S, H_1, \ldots, H_\ell) \xrightarrow{p.a,G} (b', S', H'_1, \ldots, H'_{\ell'})$ where $p$ is the priority, and such that $S \xrightarrow{a,G} S'$ is a transition of the support arena, and
  1. $(H'_1, \ldots, H'_{\ell'})$ is the tracking list obtained by updating the tracking list $(H_1, \ldots, H_\ell)$ with $G$, as explained in subsection 4.2;
  2. if $b = 1$ or if $S' \subseteq F$, then $p = 1$ and $b' = 1$;
  3. otherwise $b' = 0$. In order to compute the priority $p$, we let $p'$ be the smallest level $1 \leq r \leq \ell$ such that $H_r$ leaks at $G$ and $p' = \ell + 1$ if there is no such level, and we also let $p''$ as the minimal level $1 \leq r \leq \ell$ such that $H'_r \neq H_r \cdot G$ and $p'' = \ell + 1$ if there is no such level. Then $p = \min(2p' + 1, 2p'')$.

We are ready to state the main result of this paper, which yields an EXPTIME complexity for the population control problem. This entails the first statement of Theorem 4, and together with Proposition 13, also the first statement of Theorem 5.

▶ **Theorem 18.** Controller wins the game $PG$ if and only if Controller wins the capacity game. Solving these games can be done in time $O(2^{Q^3 + |Q|^2 + 2}(2|Q|^2 + 2))$. Strategies with $2^{|Q|^2}$ memory states are sufficient to both Controller and Agents.

**Proof.** The state space of parity game $PG$ is the product of the set of supports with a deterministic automaton computing the tracking list. There is a natural correspondence between plays and strategies in the parity game $PG$ and in the capacity game.

Controller can win the parity game $PG$ in two ways: either the play visits the support $\{f\}$, or the priority of the play is $2r + 1$ for some level $1 \leq r \leq |Q|^2$. By design of $PG$, this second possibility occurs if $r$ is remanent and leaks infinitely often. According to Lemma 17, this occurs if and only if the corresponding play of the capacity game has infinite capacity. Thus Controller wins $PG$ iff he wins the capacity game.

In the parity game $PG$, there are at most $2^{1+|Q|}\left(2^{|Q|^2}\right)^{|Q|^2} = 2^{1+|Q|^2+|Q|^4}$ states and $2^{|Q|^2} + 2$ priorities, implying the complexity bound using state-of-the-art algorithms [18]. Actually the complexity is even quasi-polynomial according to the algorithms in [10]. Notice however that this has little impact on the complexity of the population control problem, as the number of priorities is logarithmic in the number of states of our parity game.

Further, it is well known that the winner of a parity game has a positional winning strategy [18]. A *positional* winning strategy $\sigma$ in the game $PG$ corresponds to a *finite-memory* winning strategy $\sigma'$ in the capacity game, whose memory states are the states of $PG$. Actually in order to play $\sigma'$, it is enough to remember the tracking list, i.e. the third component of the state space of $PG$. Indeed, the second component, in $2^Q$, is redundant with the actual state of the capacity game and the bit in the first component is set to 1 when the play visits $\{f\}$ but in this case the capacity game is won by Controller whatever is played afterwards. Since there at most $2^{|Q|^2}$ different tracking lists, we get the upper bound on the memory. ▶
5 Lower bounds

The proofs of Theorems 4 and 5 are concluded by the proofs of lower bounds.

Theorem 19. The population control problem is EXPTIME-hard.

Proof. We first prove PSPACE-hardness of the population control problem, reducing from the halting problem for polynomial space Turing machines. We then extend the result to obtain the EXPTIME-hardness, by reducing from the halting problem for polynomial space alternating Turing machines. Let $M = (S, \Gamma, \Sigma, s_0, s_f)$ be a Turing machine with $\Gamma = \{0, 1\}$ as tape alphabet. By assumption, there exists a polynomial $P$ such that, on initial configuration $x \in \{0, 1\}^n$, $M$ uses at most $P(n)$ tape cells. A transition $t \in T$ is of the form $t = (s, s', b, b', d)$, where $s$ and $s'$ are, respectively, the source and the target control states, $b$ and $b'$ are, respectively, the symbols read from and written on the tape, and $d \in \{\leftarrow, \rightarrow, -\}$ indicates the move of the tape head. From $M$ and $x$, we build an NFA $A = (Q, \Sigma, q_0, \Delta)$ with a distinguished state $\emptyset$ such that, $M$ terminates in $s_f$ on input $x$ if and only if $(A, \emptyset)$ is a positive instance of the population control problem.

The high-level description of $A$ is as follows. States in $Q$ are of several types: contents of the $P(n)$ cells (one state $(b, p)$ per content and per position), position of the tape head (one state $p$ per possible position), control state of the Turing machine (one state $s$ per control state), and three special states, namely an initial state $q_0$, a sink winning state $\emptyset$, and a sink losing state $\emptyset$. With each transition $t = (s, s', b, b', d)$ in the Turing machine and each position $p$ of the tape, we associate an action $a_{t,p}$ in $A$, which simulates the effect of transition $t$ when the head position is $p$. Thus, on action $a_{t,p}$ there is a transition from the source state $s$ to the target state $s'$, another from the tape head position $p$ to its update according to $d$, and also from $(b, p)$ to $(b', p)$. Moreover, from head position $q \neq p$, $a_{t,p}$ leads to $\emptyset$, so that in any population game, Controller only plays actions associated with the current head position. Similarly from states $(b', p)$ with $b' \neq b$, states $s' \neq s$, action $a_{t,p}$ leads to $\emptyset$. Initially, an init action is available from $q_0$ and leads to $s_0$, to position $0$ for the tape head, and to cells $(b, p)$ that encode the initial tape contents on input $x$. The NFA also has winning actions, that allow one to check that there are no agents in a subset of states, and send the remaining ones to the target $\emptyset$. One such action should be played when agents encoding the state of the Turing machine lie in $s_f$, indicating that $M$ accepted. Another winning action win is played whenever there are not enough agents to encode the initial configuration: Agents needs $m$ to be at least $P(n) + 2$ to fill states corresponding to the initial tape contents ($P(n)$ tokens), the initial control state $s_0$ and the initial head position. The sink losing state $\emptyset$ is used to pinpoint an error in the simulation of $M$.

Now, in order to encode an alternating Turing machine, we assume that the control states of $M$ alternate between states of Controller and states of Agents. The NFA $A$ is extended with a state $C$, for Controller, and an additional transition labelled init from $q_0$ to $C$. Assume first, that $C$ contains at most an agent; we will later explain how to impose this. Beyond $C$, the NFA also contains on state $t$ per transition of $M$, which will represent that Agents chooses to play transition $t$. To do so, from state $C$, for any action $a_{t,p}$, there are transitions to all states $t'$. From state $t$, actions of the form $a_{t,p}$ are allowed, leading back to $C$. That is, actions $a_{t', p}$ with $t' \neq t$ lead from $t$ to the sink losing state $\emptyset$. This encodes that Controller must follow the transition $t$ chosen by Agents. To punish Agents in case the current tape contents is not the one expected by the transition $t = (s, s', b, b', d)$ he chooses, there are trashning actions trash, and trash$_{p,b}$ enabled from state $t$. Action trash leads from $t$ to $\emptyset$, and also from $s$ to $\emptyset$. Similarly, trash$_{p,b}$ leads from $t$ to $\emptyset$ and from any position state $q \neq p$ to $\emptyset$, and from $(b, p)$ to $\emptyset$. In this way, Agents will not move the token
from $C$ to an undesired $t$. Last, there are transitions on action \texttt{end} from state $\otimes$, $C$ and any of the $t$'s to the target state $\otimes$. Moreover, action \texttt{end} from any other state (in particular the ones encoding the Turing machine configuration) leads to $\otimes$. This whole construction encodes, assuming that there is a single agent in $C$ after the first transition, that Controller can choose the transition from a Controller state of $\mathcal{M}$, and Agents can choose the transition from an Agents state.

Let us now explain how to deal with the case where Agents places several agents in state $C$ on the initial action \texttt{init}, enabling the possibility to later send agents to several $t$'s simultaneously. For that, consider the gadget from Figure 6. We use an extra state $s$, actions $\texttt{store}_t$ for each transition $t$, and action \texttt{restart}. Action $\texttt{store}_t$ leads from $t$ to $\texttt{store}$, and loops on every other state. From all states except $\otimes$ and $\otimes$, action \texttt{restart} leads to $q_0$. Last, the effects of \texttt{win} and \texttt{end} are modified as follow: \texttt{win} leads from (non winning control states) $s \neq s_f$ to $\otimes$ and loops on every other Turing machine, including $s_f$. It also leads from $C$ and from any $t$ to $\otimes$; \texttt{end} goes from $q_0$, $C$, the $t$'s and $s$ to $\otimes$ (it can be played only if all tokens from Figure 6 are in $\otimes$), and leads all tokens from the Turing machine configuration to $\otimes$.

Assume that input $x$ is not accepted by the alternating Turing machine $\mathcal{M}$, and let $m$ be at least $P(n) + 3$. In the $m$-population game, Agents has a winning strategy placing initially a single agent in state $C$. If Controller plays \texttt{store}_t (for some $t$), either no agents are stored, or the unique agent in $C$ is moved to $s$. Thus Controller cannot play \texttt{end} and has no way to lead the agents encoding the Turing machine configuration to $\otimes$, until he plays \texttt{restart}, which moves all the agents back to $q_0$. This shows that $\texttt{store}_t$ is useless to Controller and thus Agents wins.

Conversely, if Controller has a strategy in $\mathcal{M}$ witnessing the acceptance of $x$, in order to win the $m$-population game, Agents would need to cheat in the simulation of $\mathcal{M}$ and place at least two agents in $C$ to eventually split them to $t_1, \ldots, t_n$. Then, Controller can play the corresponding actions $\texttt{store}_{t_1}, \ldots, \texttt{store}_{t_n}$ moving all agents (but the ones in $t_1$) in $s$, after which he plays his winning strategy from $t_1$ resulting in sending some agents to $\otimes$. Then, Controller plays \texttt{restart} and proceeds inductively with strictly less agents from $q_0$, and eventually plays \texttt{end} to win.

Surprisingly, the cut-off can be as high as doubly exponential in the size of the NFA.

\begin{proposition}
There exists a family of NFA $(A_n)_{n \in \mathbb{N}}$ such that $|A_n| = 2n + 7$, and for $M = 2^{2^n} + n$, there is no winning strategy in $A_n^M$ and there is one in $A_n^{M-1}$.
\end{proposition}

\textbf{Proof.} Let $n \in \mathbb{N}$. The NFA $A_n$ we build is the disjoin union of two NFAs with different properties, namely $A_{\text{split}}, A_{\text{count}}, n$. On the one hand, for $A_{\text{split}}$, winning the game with $m$
agents requires $\Theta(\log m)$ steps. On the other hand, $A_{\text{count},n}$ implements a usual counter over $n$ bits (as used in many different publications), such that Controller can avoid to lose during $O(2^n)$ steps. The combination of these two gadgets ensures a cut-off for $A_n$ of $2^n$.

Recall Figure 1, which presents the splitting gadget that has the following properties. In $A_{\text{split}}^n$ with $m \in \mathbb{N}$ agents, (s1) there is a winning strategy ensuring to win in $2 \lceil \log_2 m \rceil + 2$ steps; (s2) no strategy can ensure to win in less than $2 \lceil \log_2 m \rceil + 1$ steps.

The counting gadget that implements a counter with states $\ell_i$ (meaning bit $i$ is 0) and $h_i$ (for bit $i$ is 1) enjoys the following properties: (c1) there is a strategy in $A_{\text{count},n}$ to ensure avoiding $\ominus$ during $2^n$ steps, by playing $\alpha_i$ whenever the counter suffix from bit $i$ is $01\cdots 1$; (c2) for $m \geq n$, no strategy of $A_{\text{count},n}$ avoid $\ominus$ for $2^n$ steps.

The two gadgets (splitting and counting) are combined by a new initial state leading by two transitions labeled $\textit{init}$ to the initial states of both NFAs. Actions consist of pairs of actions, one for each gadget: $\Sigma = \{a, b, \delta\} \times \{\alpha_i \mid 1 \leq i \leq n\}$. We add an action $\ast$ which can be played from any state of $A_{\text{count},n}$ but $\ominus$, and only from $f$ in $A_{\text{split}}$, leading to the global target state $\ominus$.

Let $M = 2^{n+1} + n$. We deduce that the cut-off is $M - 1$ as follows:

- For $M$ agents, a winning strategy for Agents is to first split $n$ tokens from the initial state to the $q_0$ of $A_{\text{count},n}$, in order to fill each $\ell_i$ with 1 token, and $2^{n+1}$ tokens to the $q_0$ of $A_{\text{split}}$. Then Agents splits evenly tokens between $q_1, q_2$ in $A_{\text{split}}$. In this way, Controller needs at least $2^n + 1$ steps to reach the final state of $A_{\text{split}}$ (s2), but Controller reaches $\ominus$ after these $2^n + 1$ steps in $A_{\text{count},n}$ (c2).

- For $M - 1$ agents, Agents needs to use at least $n$ tokens from the initial state to the $q_0$ of $A_{\text{count},n}$, else Controller can win easily. But then there are less than $2^{n+1}$ tokens in the $q_0$ of $A_{\text{split}}$. And thus by (s1), Controller can reach $f$ within $2^n$ steps, after which he still avoids $\ominus$ in $A_{\text{count},n}$ (c1). And then Controller sends all agents to $\ominus$ using $\ast$.

Thus, the family $(A_n)$ of NFA exhibits a doubly exponential cut-off.

6 Discussion

Obtaining an EXPTIME algorithm for the control problem of a population of agents was challenging. We also managed to prove a matching lower-bound. Further, the surprising doubly exponential matching upper and lower bounds on the cut-off imply that the alternative technique, checking that Controller wins all $m$-population game for $m$ up to the cut-off, is far from being efficient.

The idealised formalism we describe in this paper is not entirely satisfactory: for instance, while each agent can move in a non-deterministic way, unrealistic behaviours can happen, e.g. all agents synchronously taking infinitely often the same choice. An almost-sure control problem in a probabilistic formalism should be studied, ruling out such extreme behaviours. As the population is discrete, we may avoid the undecidability that holds for distributions [11] and is inherited from the equivalence with probabilistic automata [17]. Abstracting continuous distributions by a discrete population of arbitrary size could thus be seen as an approximation technique for undecidable formalisms such as probabilistic automata.

Acknowledgements. We are grateful to Gregory Batt for fruitful discussions concerning the biological setting. Thanks to Mahsa Shirmohammadi for interesting discussions.
References


The Robot Routing Problem for Collecting Aggregate Stochastic Rewards

Rayna Dimitrova\textsuperscript{1}, Ivan Gavran\textsuperscript{2}, Rupak Majumdar\textsuperscript{3}, Vinayak S. Prabhu\textsuperscript{4}, and Sadegh Esmaeil Zadeh Soudjani\textsuperscript{5}

\textbf{Abstract}

We propose a new model for formalizing reward collection problems on graphs with dynamically generated rewards which may appear and disappear based on a stochastic model. The robot routing problem is modeled as a graph whose nodes are stochastic processes generating potential rewards over discrete time. The rewards are generated according to the stochastic process, but at each step, an existing reward disappears with a given probability. The edges in the graph encode the (unit-distance) paths between the rewards’ locations. On visiting a node, the robot collects the accumulated reward at the node at that time, but traveling between the nodes takes time. The optimization question asks to compute an optimal (or $\epsilon$-optimal) path that maximizes the expected collected rewards.

We consider the finite and infinite-horizon robot routing problems. For finite-horizon, the goal is to maximize the total expected reward, while for infinite horizon we consider limit-average objectives. We study the computational and strategy complexity of these problems, establish NP-lower bounds and show that optimal strategies require memory in general. We also provide an algorithm for computing $\epsilon$-optimal infinite paths for arbitrary $\epsilon > 0$.

1 Introduction

Reward collecting problems on metric spaces are at the core of many applications, and studied classically in combinatorial optimization under many well-known monikers: the traveling salesman problem, the knapsack problem, the vehicle routing problem, the orienteering problem, and so on. Typically, these problems model the metric space as a discrete graph whose nodes or edges constitute rewards, either deterministic or stochastic, and ask how to traverse the graph to maximize the collected rewards. In most versions of the problem,
rewards are either fixed or cumulative. In particular, once a reward appears, it stays there until collection. However, in many applications, existing rewards may disappear (e.g., a customer changing her mind) or have more “value” if they are collected fast.

We introduce the Robot Routing problem, which combines the spatial aspects of traveling salesman and other reward collecting problems on graphs with stochastic reward generation and with the possibility that uncollected rewards may disappear at each stage. The robot routing problem consists of a finite graph and a reward process for each node of the graph. The reward process models dynamic requests which appear and disappear. At each (discrete) time point, a new reward is generated for the node according to a stochastic process with expectation $\lambda$. However, at each point, a previously generated reward disappears with a fixed probability $\delta$. When the node is visited, the entire reward is collected. The optimization problem for robot routing asks, given a graph and a reward process, what is the optimal (or $\epsilon$-optimal) path a robot should traverse in this graph to maximize the expected reward?

As an illustrating example for our setting, consider a vendor planning her path through a city. At each street corner, and at each time step, a new customer arrives with expectation $\lambda$, and an existing customer leaves with probability $\delta$. When the vendor arrives at the corner, she serves all the existing requests at once. We ignore other possible real-world features and behaviors e.g., customers leaving queues if the queue length is long. How should the vendor plan her path? Similar problems can be formulated for traffic pooling [25], for robot control [13], for patrolling [15], and many other scenarios.

Despite the usefulness of robot routing in many scenarios involving dynamic appearance and disappearance of rewards, algorithms for its solution have not, to the best of our knowledge, been studied before. In this paper, we study two optimization problems: the value computation problem, that asks for the maximal expected reward over a finite or infinite horizon, and the path computation problem, that asks for a path realizing the optimal (or $\epsilon$-optimal) reward. The key observation to solving these problems is that the reward collection can be formulated as discounted sum problems over an extended graph, using the correspondence between stopping processes and discounted sum games.

For finite horizon robot routing we show that the value decision problem (deciding if the maximal expected reward is at least a certain amount) is NP-complete when the horizon bound is given in unary, and the value and optimal path can be computed in exponential time using dynamic programming.

For the infinite horizon problem, where the accumulated reward is defined as the long run average, we show that the value decision problem is NP-hard if the probability of a reward disappearing is more than a threshold dependent on the number of nodes. We show that computing the optimal long run average reward can be reduced to a 1-player mean-payoff game on an infinite graph. By solving the mean payoff game on a finite truncation of this graph, we can approximate the solution up to an arbitrary precision. This gives us an algorithm that, for any given $\epsilon$, computes an $\epsilon$-optimal path in time exponential in the size of the original graph and logarithmic in $1/\epsilon$. Unlike finite mean-payoff 2-player games, strategies which generate optimal paths for robot routing even in the 1-player setting can require memory. For the non-discounted infinite horizon problem (that is, when rewards do not disappear) we show that the optimal path and value problems are solvable in polynomial time.

Related work. The robot routing problem is similar in nature to a number of other problems studied in robot navigation, vehicle routing, patrolling, and queueing network control, but to the best of our knowledge has not been studied so far.

There exists a plethora of versions of the famous traveling salesman problem (TSP) which explore the trade-off between the cost of the constructed path and its reward. Notable
examples include the orienteering problem [23], in which the number of locations visited in a limited amount of time is to be maximized, vehicle routing with time-windows [17] and deadlines-TSP [2], which impose restrictions or deadlines on when locations should be visited, as well as discounted-reward-TSP [4] in which soft deadlines are implemented by means of discounting. Unlike in our setting, in all these problems, rewards are static, and there is no generation and accumulation of rewards, which is a key feature of our model.

In the dynamic version of vehicle routing [7] and the dynamic traveling repairman problem [3], tasks are dynamically introduced and the objective is to minimize the expected task waiting time. In contrast, we focus on limit-average objectives, which are a classical way to combine rewards over infinite system runs. Patrolling [6] is another graph optimization problem, motivated by operational security. The typical goal in patrolling is to synthesize a strategy for a defender against a single attack at an undetermined time and location, and is thus incomparable to ours. A single-robot multiple-intruders patrolling setting that is close to ours is described in [15], but there again the objective is to merely detect whether there is a visitor/intruder at a given room. Thus, the patrolling environment in [15] is described by the probability of detecting a visitor for each location. On the contrary, our model can capture counting patrolling problems, where the robot is required not only to detect the presence of visitors but to register/count as many of them as possible. Another related problem is the information gathering problem [20]. The key difference between the information gathering setting and ours is that [20] assumes that making an observation earlier has bigger value than if a lot of observations have already been made. This restriction on the reward function is not present in our model, since the reward value collected when visiting node $v$ at time $t$ (making observation $(v,t)$, in their terms) only depends on the last time when $v$ was previously visited, and not on the rest of the path (the other observations made, in their terms).

Average-energy games [8, 5] are a class of games on finite graphs in which the limit-average objective is defined by a double summation. The setting discussed in [8, 5] considers static edge weights and no discounting. Moreover, the inner sum in an average-energy objective is over the whole prefix so far, while in our setting the inner sum spans from the last to the current visit of the current node, which is a crucial difference between these two settings.

Finally, there is a rich body of work on multi-robot routing [24, 1, 18, 10, 11] which is closely related to our setting. However, the approaches developed there are limited to static tasks with fixed or linearly decreasing rewards. The main focus in the multi-robot setting is the task allocation and coordination between robots, which is a dimension orthogonal to the aggregate reward collection problem which we study.

Markov decision processes (MDP) [19] seem superficially close to our model. In an MDP, the rewards are determined statically as a function of the state and action. In contrast, the dynamic generation and accumulation of rewards in our model, especially the individual discounting of each generated reward, leads to algorithmic differences: for example, while MDPs admit memoryless strategies for long run average objectives, strategies require memory in our setting and there is no obvious reduction to, e.g., an exponentially larger, MDP.

We employed the reward structure of this article in [13] with the goal of synthesizing controllers for reward collecting Markov processes in continuous space. The work [13] is mainly focused on addressing the continuous dynamics of the underlying Markov process where the authors use abstraction techniques [12] to provide approximately optimal controllers with formal guarantees on the performance while maintaining the probabilistic nature of the process. In contrast, we tackle the challenges of this problem with having a deterministic graph as the underlying dynamical model of the robot and study the computational complexity of the proposed algorithms thoroughly.
Contributions. We define a novel optimization problem for formalizing and solving reward collection in a metric space where stochastic rewards appear as well as disappear over time.

- We consider reward-collection problems in a novel model with dynamic generation and accumulation of rewards, where each reward can disappear with a given probability.
- We study the value decision problem, the value computation problem, and the path computation problem over a finite horizon. We show that the value decision problem is NP-complete when the horizon is given in unary. We describe a dynamic programming approach for computing the optimal value and an optimal path in exponential time.
- We study the value decision problem, the value computation problem, and the path computation problem over an infinite horizon. We show that for sufficiently large values of the disappearing factor δ the value decision problem is NP-hard. We provide an algorithm which for any given ϵ > 0, computes an ϵ-optimal path in time exponential in the size of the original graph and logarithmic in 1/ϵ. We demonstrate that strategies (in the 1-player robot routing games) which generate infinite-horizon optimal paths can require memory.

2 Problem Formulation

Preliminaries and notation. A finite directed graph $G = (V, E)$ consists of a finite set of nodes $V$ and a set of edges $E \subseteq V \times V$. A path $\pi = v_0, v_1, \ldots$ in $G$ is a finite or infinite sequence of nodes in $G$, such that $(v_i, v_{i+1}) \in E$ for each $i \geq 0$. We denote with $|\pi| = N$ the length (number of edges) of a finite path $\pi = v_0, v_1, \ldots, v_N$ and write $\pi[i] = v_i$ and $\pi[0] \ldots N] = v_0, v_1, \ldots, v_N$. For an infinite path $\pi$, we define $|\pi| = \infty$. We also denote the cardinality of a finite set $U$ by $|U|$. We denote by $\mathbb{N} = \{0, 1, \ldots\}$ and $\mathbb{Z}_+ = \{1, 2, \ldots\}$ the sets of non-negative and positive integers respectively. We define $\mathbb{Z}[n, m] = \{n, n+1, \ldots, m\}$ for any $n, m \in \mathbb{N}, n \leq m$. We denote with $\mathbb{I}(\cdot)$ the indicator function which takes a Boolean-valued expression as its argument and returns 1 if this expression evaluates to true and 0 otherwise.

Problem setting. Fix a graph $G = (V, E)$. We consider a discrete-time setting where at each time step $t \in \mathbb{N}$, at each node $v \in V$, a reward process generates rewards according to some probability distribution. Once generated, each reward at a node decays according to a decaying function. A reward-collecting robot starts out at some node $v_0 \in V$ at time $t = 0$, and traverses one edge in $E$ at each time step. Every time the robot arrives at a node $v \in V$, it collects the reward accumulated at $v$ since the last visit to $v$. Our goal is to compute the maximum expected reward that the robot can possibly collect, and to construct an optimal path for the robot in the graph, i.e., a path whose expected total reward is maximal.

To formalize reward accumulation, we define a function $\text{Last}_\pi$ which (for path $\pi$) maps an index $t \leq |\pi|$ and a node $v \in V$ to the length of the path starting at the previous occurrence of $v$ in $\pi$ till position $t$; and to $t + 1$ if $v$ does not occur in $\pi$ before time $t$:

$$\text{Last}_\pi(t,v) := \min(t + 1, \{t - j \in \mathbb{N} \mid j < t, \pi[j] = v\}).$$

Reward functions. Let $\xi : \Omega \times V \to \mathbb{R}$ be a set of random variables defined on a sample space $\Omega$ and indexed by the set of nodes $V$. Then $\xi(\cdot, v), v \in V$, is a measurable function from $\Omega$ to $\mathbb{R}$ that generates a random reward at node $v$ at any time step. Let $\pi$ be the path in $G$ traversed by the robot. At time $t$, the position of the robot is the node $\pi[t]$, and the robot collects the uncollected decayed reward generated at node $\pi[t]$ (since its last visit to $\pi[t]$) up till and including time $t$. Then, the robot traverses the edge $(\pi[t], \pi[t + 1])$, and at time $t + 1$ it collects the rewards at node $\pi[t + 1]$. 

---

The Robot Routing Problem for Collecting Aggregate Stochastic Rewards
The uncollected reward at time $t$ at a node $v$ given a path $\pi$ traversed by the robot is defined by the random variable

$$\text{acc}_\pi(t, v) := \sum_{j=0}^{\text{Last}_\pi(t, v) - 1} \gamma(v)^j \xi(w(t - j), v), \quad w(\cdot) \in \Omega.$$  

The value $\gamma(v)$ in the above definition is a discounting factor that models the probability that a reward at node $v$ survives for one more round, that is, the probability that a given reward instance at node $v$ disappears at any step is $1 - \gamma(v)$.

Note that the previous time a reward was collected at node $v$ was at time $t - \text{Last}_\pi(v, t)$, the time node $v$ was last visited before $t$. Thus $\text{acc}_\pi(t, v)$ corresponds to the rewards generated at node $v$ at times $t, t - 1, \ldots, t - \text{Last}_\pi(t, v) + 1$, which have decayed by factors of $\gamma(v)^0, \gamma(v)^1, \ldots, \gamma(v)^{\text{Last}_\pi(t, v) - 1}$, respectively. When traversing a path $\pi$, the robot collects the accumulated reward $\text{acc}_\pi(t, \pi[t])$ at time $t$ at node $\pi[t]$.

We define the expected finite $N$-horizon sum reward for a path $\pi$ as:

$$r_{\text{sum}}^{(N)}(\pi) := \mathbb{E} \left[ \sum_{t=0}^{N} \text{acc}_\pi(t, \pi[t]) \right].$$  

Let $\lambda : V \to \mathbb{R}_{\geq 0}$ be a function that maps each node $v \in V$ to the expected value of the reward generated at node $v$ for each time step, $\lambda(v) = \mathbb{E}[\xi(\cdot, v)]$. We assume that the rewards generated at each node are independent of the agent’s move. Thus, the function $\lambda$ will be sufficient for our study, since we have

$$r_{\text{sum}}^{(N)}(\pi) = \sum_{t=0}^{N} \mathbb{E}\text{acc}_\pi(t, \pi[t]), \text{where } \mathbb{E}\text{acc}_\pi(t, v) := \sum_{j=0}^{\text{Last}_\pi(t, v) - 1} \gamma(v)^j \lambda(v). \quad (1)$$  

For an infinite path $\pi$, the limit-average expected reward is defined as

$$r_{\text{av}}(\pi) = \liminf_{N \to \infty} \frac{r_{\text{sum}}^{(N)}(\pi)}{N + 1}. \quad (2)$$  

The finite and infinite-horizon reward values for a node $v$ are defined as the best rewards over all paths originating in $v$: $R_{\text{sum}}^{(N)}(v) = \sup_\pi \left\{ r_{\text{sum}}^{(N)}(\pi) \mid \pi[0] = v, |\pi| = N \right\}$ and $R_{\text{av}}(v) = \sup_\pi \left\{ r_{\text{av}}(\pi) \mid \pi[0] = v, |\pi| = \infty \right\}$, respectively. The choice of limit-average in (2) is due to the unbounded sum reward $r_{\text{sum}}^{(N)}(\pi)$ when $N$ goes to infinity. For a given path $\pi$, the sequence $r_{\text{sum}}^{(N)}(\pi)/(N + 1)$ in (2) may not converge. Thus we opt for the worst case limiting behavior of the sequence. Alternatively, one may select the best case limiting behavior $\lim\sup_{N \to \infty}$ in (2) with no substantial change in the results of this paper.

**Node-invariant functions $\lambda$ and $\gamma$ and definition of cost functions.** In the case when the functions $\lambda$ and $\gamma$ are constant, we write $\lambda$ and $\gamma$ for the respective constants. In this case, the expressions for $r_{\text{sum}}^{(N)}(\pi)$ and $r_{\text{av}}(\pi)$ can be simplified using the identity $1 + \gamma + \gamma^2 + \cdots + \gamma^{q-1} = \frac{1 - \gamma^q}{1 - \gamma}$ for $\gamma < 1$. Then we have

$$r_{\text{sum}}^{(N)}(\pi) = \lambda \sum_{t=0}^{N} \sum_{j=0}^{\text{Last}_\pi(t, \pi[t]) - 1} \gamma^j = \lambda \sum_{t=0}^{N} \frac{1 - \gamma^{\text{Last}_\pi(t, \pi[t])}}{1 - \gamma} = \frac{(N + 1)\lambda}{1 - \gamma} - \frac{\lambda}{1 - \gamma} \sum_{t=0}^{N} \gamma^{\text{Last}_\pi(t, \pi[t])}. \quad (3)$$
Thus we have $\pi$ as a running example throughout the paper. The functions $r$ and $C$ correspond to computing paths that minimize the cost functions $r_{\text{sum}}$ and $C_{\text{av}}$ respectively. Analogously to $R_{\text{sum}}$ and $R_{\text{av}}$, the infimums of the cost functions in (5) over paths are denoted by $C_{\text{sum}}(v)$ and $C_{\text{av}}(v)$ respectively.

\textbf{Example 1.} Consider the graph $G_e = (V_e, E_e)$ in Figure 1 with $V_e = \{a, b, c, d\}$, which we will use as a running example throughout the paper. The functions $\lambda$ and $\gamma$ are constant.

Consider the finite path $\pi_1 = \text{adabc}$. For the occurrences of node $a$ in $\pi_1$ we have $\text{Last}_{\pi_1}(0, a) = 1$, $\text{Last}_{\pi_1}(2, a) = 2$, $\text{Last}_{\pi_1}(5, a) = 3$, and similarly for the other nodes in $\pi_1$. The reward for $\pi_1$ as a function of $\lambda$ and $\gamma$ is $r_{\text{sum}}(\pi_1) = \frac{\lambda}{\gamma^3} - \frac{\lambda}{\gamma^2} \left( \gamma + \gamma^2 + \gamma^4 + \gamma^5 + \gamma^3 + \gamma^5 \right)$ for $\gamma < 1$ and $r_{\text{sum}}(\pi_1) = 22\lambda$ for $\gamma = 1$. For the infinite path $\pi_2 = \text{abc}$ we have $\text{Last}_{\pi_2}(0, a) = 1$, $\text{Last}_{\pi_2}(1, b) = 2$, $\text{Last}_{\pi_2}(2, c) = 3$ and the value of $\text{Last}$ is 3 in all other cases. Thus we have $r_{\text{av}}(\pi_2) = \frac{\lambda}{\gamma^3} - \frac{\lambda}{\gamma^2} \gamma^3$ for $\gamma < 1$ and $r_{\text{av}}(\pi_2) = 3\lambda$ for $\gamma = 1$. Similarly, for $\pi_3 = \text{adabc}$ we have $r_{\text{av}}(\pi_3) = \frac{\lambda}{\gamma^3} - \frac{\lambda}{\gamma^2} \left( \gamma^2 + \gamma^3 + \gamma^5 + \gamma^3 + \gamma^5 \right)$ for $\gamma < 1$ and $r_{\text{av}}(\pi_3) = 4\lambda$ for $\gamma = 1$.

\textbf{Problem statements.} We investigate optimization and decision problems for finite and infinite-horizon robot routing. The value computation problems ask for the computation of $R_{\text{sum}}(v)$ and $R_{\text{av}}(v)$. The corresponding decision problems asks to check if the respective one of these two quantities is greater than or equal to a given threshold $R \in \mathbb{R}$.

\textbf{Definition 2 (Value Decision Problems).} Given a finite directed graph $G = (V, E)$, an expected reward function $\lambda : V \to \mathbb{R}_{\geq 0}$, a discounting function $\gamma : V \to (0, 1]$, an initial node $v_0 \in V$ and a threshold value $R \in \mathbb{R}$,

\begin{itemize}
  \item The finite horizon value decision problem is to decide, given $N$, if $R_{\text{sum}}(v_0) \geq R$.
  \item The infinite horizon value decision problem is to decide if $R_{\text{av}}(v_0) \geq R$.
\end{itemize}

For a finite directed graph $G = (V, E)$, expected reward and discounting functions $\lambda : V \to \mathbb{R}_{\geq 0}$ and $\gamma : V \to (0, 1]$ and $v_0 \in V$, a finite path $\pi$ is said to be an optimal path for time-horizon $N$ if (a) $\pi[0] = v_0$ and $|\pi| = N$, and (b) for every path $\pi'$ in $G$ with $\pi'[0] = v_0$.
and $|\pi'| = N$ it holds that $r^{(N)}_{\text{sum}}(\pi) \geq r^{(N)}_{\text{sum}}(\pi')$. Similarly, an infinite path $\pi$ is said to be optimal for the infinite horizon if $\pi[0] = v_0$ and for every infinite path $\pi'$ with $\pi'[0] = v_0$ in $G$ it holds that $r_{\text{av}}(\pi) \geq r_{\text{av}}(\pi')$. We can also define corresponding threshold paths: given a value $R$ a path $\pi$ is said to be threshold $R$-optimal if $r^{(N)}_{\text{sum}}(\pi) \geq R$ or $r_{\text{av}}(\pi) \geq R$, respectively. An $\epsilon$-optimal path is one which is $R^{(N)}_{\text{sum}}(v_0) - \epsilon$ or $R_{\text{av}}(v_0) - \epsilon$ threshold optimal (for finite or infinite horizon respectively).

Example 3. Consider again the graph $G_e$ shown in Figure 1. Examining the expressions computed in Example 1, we have $r_{\text{av}}(\pi_2) > r_{\text{av}}(\pi_3)$ for $\gamma = 0.1$ and $r_{\text{av}}(\pi_3) > r_{\text{av}}(\pi_2)$ for $\gamma = 0.9$. Thus, in general, the optimal value depends on $\gamma$. Due to the structure of the set of infinite paths in $G_e$ we can analytically compute the optimal value $R_{\text{av}}(v)$ for each $v \in V_e$ as a function of $\gamma$ and a corresponding optimal path (see the full version [9] for the proof):

- if $\gamma \in [a_1, a_2]$, then $R_{\text{av}}(v) = \frac{\lambda}{1-\gamma} \frac{1-\gamma^3}{1-\gamma^4}$ and the path $(abc)^\omega$ is optimal;
- if $\gamma \in [a_2, 1]$, then $R_{\text{av}}(v) = \frac{\lambda}{1-\gamma} \frac{1-\gamma^3}{1-\gamma^4}$ and the path $(abcd)^\omega$ is optimal.

The constants $a_1 \approx 0.2587$ and $a_2 \approx 0.2738$ are respectively the unique real roots of polynomials $\gamma^6 + 2\gamma^5 - 4\gamma + 1$ and $5\gamma^6 - 14\gamma^3 + 12\gamma - 3$ in the interval $(0, 1)$. Note that for $\gamma = 1$ we have $R_{\text{av}}(v) = 4\lambda$ which is achieved by $(abcd)^\omega$. The path $(abc)(ad)(abc)(2)(ad)^2 \ldots (abc)^n(ad)^n \ldots$, which is not ultimately periodic, also achieves the optimal reward.

Paths as strategies. We often refer to infinite paths as resulting from strategies (of the collecting agent). A strategy $\sigma$ in $G$ is a function that maps finite paths $\pi[0 \ldots m]$ to nodes such that if $\sigma(\pi)$ is defined then $(\pi[m], \sigma(\pi)) \in E$. Given an initial node $v_0$, the strategy $\sigma$ generates a unique infinite path $\pi$, denoted as outcome($v_0, \sigma$). Thus, every infinite path $\pi = v_0, v_1, \ldots$ defines a unique strategy $\sigma_\pi$ where $\sigma_\pi(\pi[0 \ldots i]) = v_{i+1}$, and $\sigma_\pi(e) = v_0$, and $\sigma_\pi$ is undefined otherwise. Clearly, outcome($v_0, \sigma_\pi$) = $\pi$. We say a strategy $\sigma$ is optimal for a path problem if the path outcome($v_0, \sigma$) is optimal. A strategy $\sigma$ is memoryless if for every two paths $\pi[0 \ldots m], \pi'[0 \ldots m']$ for which $\pi'[m'] = \pi''[m''], \sigma(\pi') = \sigma(\pi'')$. We say that memoryless strategies suffice for the optimal path problem if there always exists a memoryless strategy $\sigma$ such that outcome($v_0, \sigma$) is an optimal path.

3 Finite Horizon Rewards: Computing $R^{(N)}_{\text{sum}}(v)$

In this section we consider the finite-horizon problems associated with our model. The following theorem summarizes the main results.

Theorem 4. Given $G = (V,E)$, expected reward and discounting functions, node $v \in V$, and horizon $N \in \mathbb{N}$:
1. The finite-horizon value decision problem is NP-complete if $N$ is in unary.
2. The value $R^{(N)}_{\text{sum}}(v)$ for $v \in V$ is computable in exponential time even if $N$ is in binary.

Analogous results hold for the related reward problem where in addition to the initial node $v$, we are also given a destination node $v_f$, and the objective is to go from $v$ to $v_f$ in at most $N$ steps while maximizing the reward.

The finite-horizon problem is NP-hard by reduction from the Hamiltonian path problem (see the full version [9]), even in the case of node-invariant $\lambda$ and $\gamma$. Membership in NP in case $N$ is in unary follows from the fact that we can guess a path of length $N$ and check that the reward for that path is at least the desired threshold value.

To prove the second part of the theorem, we construct a finite augmented weighted graph.
For simplicity, we give the proof for node-invariant $\lambda$ and $\gamma$, working with the cost functions $c_{\text{sum}}$ and $c_{av}$. The augmented graph construction in the general case is a trivial generalization by changing the weights of the nodes, and the dynamic programming algorithm used for computing the optimal cost values is easily modified to compute the corresponding reward values instead. For $\gamma < 1$ the objective is to minimize $c_{\text{sum}}^{(N)}(\pi) = \sum_{t=0}^{N} \gamma^{t} \text{Last}_x(t, \pi(t))$ and for $\gamma = 1$ the objective is to maximize $r_{\text{sum}}^{(N)}(\pi) = \lambda \sum_{t=0}^{N} \text{Last}_x(t, \pi(t))$ over paths $\pi$.

**Augmented weighted graph.** Given a finite directed graph $G = (V,E)$ we define the **augmented weighted graph** $\tilde{G} = (\tilde{V}, \tilde{E})$ which “encodes” the values $\text{Last}_x(t, v)$ for the paths in $G$ explicitly in the augmented graph node. We can assume w.l.o.g. that $V = \{ 1, 2, \ldots, |V| \}$.

- The set of nodes $\tilde{V} = V \times \mathbb{Z}^{|V|}_+$ (the set $\tilde{V}$ is infinite). A node $(v, b_1, b_2, \ldots, b_{|V|}) \in \tilde{V}$ represents the fact that the current node is $v$, and that for each node $u \in V$ the last visit to $u$ (before the current time) was $b_u$ time units before the current time.

- The weight of a node $(v, b_1, b_2, \ldots, b_{|V|}) \in \tilde{V}$ is $\gamma^{b_v}$.

- The set of edges $\tilde{E}$ consists of edges $(v, b_1, b_2, \ldots, b_{|V|}) \rightarrow (v', b_1', b_2', \ldots, b_{|V|}')$ such that $(v, v') \in E$; and $b_v' = 1$, and $b_u' = b_u + 1$ for all $u \neq v$.

Let $\pi$ be a path in $G$. In the graph $\tilde{G}$ there exists a unique path $\tilde{\pi}$ that corresponds to $\pi$:

$$\tilde{\pi} = (\pi[0], 1, 1, \ldots, 1), (\pi[1], b_1^1, b_1^1, \ldots, b_{|V|}^1), (\pi[2], b_2^2, b_2^2, \ldots, b_{|V|}^2), \ldots \quad (6)$$

starting from the node $(\pi[0], 1, 1, \ldots, 1)$ such that for all $t$ and for all $v \in V$, we have $\text{Last}_x(t, v) = b_v^t$. Dually, for each path $\tilde{\pi}$ in $\tilde{G}$ starting from $(v_0, 1, 1, \ldots, 1)$, there exists a unique path $\pi$ in $G$ from the node $v_0$ such that $\text{Last}_x(t, v) = b_v^t$ for all $t$ and $v$.

For a path $\tilde{\pi}$ in the form of (6) let

$$\tilde{c}_{\text{sum}}^{(N)}(\tilde{\pi}) := \sum_{t=0}^{N} \gamma^{b_v^t} \quad \text{and} \quad \tilde{c}_{av}(\tilde{\pi}) := \limsup_{N \rightarrow \infty} \frac{1}{N+1} \sum_{t=0}^{N} \gamma^{b_v^t}. \text{Observe that:}$$

- $\tilde{c}_{\text{sum}}^{(N)}(\tilde{\pi})$ is the sum of weights associated with the first $N+1$ nodes of $\tilde{\pi}$.

- $\tilde{c}_{av}(\tilde{\pi})$ is the limit-average of the weights associated with the nodes of $\tilde{\pi}$.

Thus, $c_{\text{sum}}^{(N)}$ and $c_{av}$ define the classical total finite sum (shortest paths) and limit average objectives on weighted (infinite) graphs [26]. Additionally, $\tilde{c}_{\text{sum}}^{(N)}(\tilde{\pi}) = c_{\text{sum}}^{(N)}(\pi)$, and $\tilde{c}_{av}(\tilde{\pi}) = c_{av}(\pi)$ where $\pi$ is the path in $G$ corresponding to the path $\tilde{\pi}$.

Now, define $\tilde{C}_{\text{sum}}^{(N)}((v_0, 1, 1, \ldots, 1))$ as the infimum of $\tilde{c}_{\text{sum}}^{(N)}(\tilde{\pi})$ over all paths $\tilde{\pi}$ with $\tilde{\pi}[0] = (v_0, 1, 1, \ldots, 1)$, and similarly for $\tilde{C}_{av}^{(N)}((v_0, 1, 1, \ldots, 1)))$. Then it is easy to see that $C_{\text{sum}}^{(N)}(v_0) = \tilde{C}_{\text{sum}}^{(N)}((v_0, 1, 1, \ldots, 1))$ and $C_{av}(v_0) = \tilde{C}_{av}(((v_0, 1, 1, \ldots, 1)))$. Thus, we can reduce the optimal path and value problems for $G$ to standard objectives in $\tilde{G}$. The major difficulty is that $\tilde{G}$ is infinite. However, note that only the first $N+1$ nodes of $\tilde{\pi}$ are relevant for the computation of $\tilde{c}_{\text{sum}}^{(N)}(\tilde{\pi})$. Thus, the value of $\tilde{C}_{\text{sum}}^{(N)}((v_0, 1, 1, \ldots, 1))$ can be computed on a finite subgraph of $\tilde{G}$, obtained by considering only the finite subset of nodes $V \times \mathbb{Z}[1, N + 1]^{|V|} \subseteq \tilde{V}$.

For $\gamma < 1$, we obtain the value $\tilde{C}_{\text{sum}}^{(N)}(\tilde{\pi})$ by a standard dynamic programming algorithm which computes the shortest path of length $N$ on this finite subgraph starting from the node $(v_0, 1, 1, \ldots, 1)$ (and keeping track of the number of steps). For $\gamma = 1$, where the objective is to maximize $r_{\text{sum}}^{(N)}(\pi) = \lambda \sum_{t=0}^{N} \text{Last}_x(t, \pi(t))$ over paths $\pi$, we proceed analogously. Note that the subgraph used for the dynamic programming computations is of exponential size in terms of the size of $G$ and the description of $N$. This gives the desired result in Theorem 4.
Infinite Horizon Rewards: Computing $R_{av}(v)$

Since we consider finite graphs, every infinite path eventually stays in some strongly connected component (SCC). Furthermore, the value of the reward function $r_{av}(\pi)$ does not change if we alter/remove a finite prefix of the path $\pi$. Thus, it suffices to restrict our attention to the SCCs of the graph: the problem of finding an optimal path from a node $v \in V$ reduces to finding the SCC that gives the highest reward among the SCCs reachable from node $v$. Therefore, we assume that the graph is strongly connected.

4.1 Hardness of Exact $R_{av}(v)$ Value Computation

Since it is sufficient for the hardness results, we consider node-invariant $\lambda$ and $\gamma$.

Insufficiency of memoryless strategies. Before we turn to the computational hardness of the value decision problem, we look at the strategy complexity of the optimal path problem and show that optimal strategies need memory.

▷ Proposition 5. Memoryless strategies do not suffice for the infinite horizon problem.

Proof. Consider Example 3. A memoryless strategy results in paths which cycle exclusively either in the left cycle, or the right cycle (as from node $a$, it prescribes a move to either only $b$ or only $d$). As shown in Example 3, the optimal path for $\gamma \geq a_1$ needs to visit both cycles. Thus, memoryless strategies do not suffice for this example. ▷

For $\omega$-regular objectives, strategies based on latest visitation records [14, 22], which depend only on the order of the last node visits (i.e., for all node pairs $v_1 \neq v_2 \in V$, whether the last visit of $v_1$ was before that of $v_2$ or vice versa) are sufficient. However, we can show that such strategies do not suffice either. To see this, recall the graph in Figure 1 for which the optimal path for $\gamma = 0.26$ is $(abcabd)\omega$. Upon visiting node $a$ this strategy chooses one of the nodes $b$ or $d$ depending on the number of visits to $b$ since the last occurrence of $d$. On the other hand, every strategy based only on the order of last visits is not able to count the number of visits to $b$ and thus, results in a path that ends up in one of $(abc)^\omega$, or $(ad)^\omega$, or $(abcd)^\omega$, which are not optimal for this $\gamma$. The proof is given in the full version [9]. It is open if finite memory strategies are sufficient for the infinite-horizon optimal path problem.

NP-Hardness of the value decision problem. To show NP-hardness of the infinite-horizon value decision problem, we first give bounds on $R_{av}(v_0)$. The following Lemma proven in the full version [9], establishes these bounds.

▷ Lemma 6. For any graph $G = (V, E)$ and any node $v_0 \in V$, $R_{av}(v_0)$ is bounded as

$$\frac{1 - \gamma^p}{1 - \gamma} \leq R_{av}(v_0) \leq \frac{1 - \gamma^{n_v}}{1 - \gamma},$$

(7)

where $n_v = |V|$ and $p$ is the length of the longest simple cycle in $G$.

▷ Corollary 7. If the graph $G = (V, E)$ contains a Hamiltonian cycle, any path $\pi = (v_1v_2\ldots v_{|V|})^\omega$, with $v_1v_2\ldots v_{|V|}v_1$ being a Hamiltonian cycle, is optimal and the optimal value of $R_{av}(v_0)$ is exactly the upper bound in (7).

The following lemma establishes a relationship between the value of optimal paths and the existence of a Hamiltonian cycle in the graph, and is useful for providing a lower bound on the computational complexity of the value decision problem.
Lemma 8. If the upper bound in (7) is achieved with a path \( \pi \) for some \( \gamma < 1/|V| \), then the graph contains a Hamiltonian cycle \( \rho \) that occurs in \( \pi \) infinitely often.

Proof. The proof is by contradiction. Suppose \( \pi \) does not visit any Hamiltonian cycle infinitely often. Then it visits each such cycle at most a finite number of times. Without loss of generality we can assume that the path doesn’t visit any such cycles, since the total number of Hamiltonian cycles is finite. We have for \( n_v = |V| \)

\[
\epsilon^{(N)}_{\text{sum}}(\pi) \geq \sum_{i=0}^{N} \gamma^{\text{Last}_{\pi}(t, \pi[il])} (\text{Last}_{\pi}(t, \pi[il]) < n_v) \geq \gamma^{n_v - 1} \sum_{i=0}^{N} \gamma^{\text{Last}_{\pi}(t, \pi[il])} (\text{Last}_{\pi}(t, \pi[il]) < n_v).
\]

Now let’s look at \( \pi_{n, n_v} = \pi[n(n + 1) \ldots (n + n_v)] \) a finite sub-path of length \( n_v \). There is at least one node repeated in \( \pi_{n, n_v} \) since the graph has \( n_v \) distinct nodes. Note that if \( \pi[n] = \pi[n + n_v] \), there must be another repetition due to the lack of Hamiltonian cycles in the path. In either case, there is an \( i_n \in [n, n + n_v] \) such that \( \text{Last}_{\pi}(i_n, \pi[i_n]) < n_v \).

\[
\epsilon^{(N)}_{\text{sum}}(\pi) \geq \gamma^{n_v - 1} \sum_{i=0}^{N} \gamma^{\text{Last}_{\pi}(t, \pi[il])} (\text{Last}_{\pi}(t, \pi[il]) < n_v) \geq \gamma^{n_v - 1} \frac{N}{n_v}.
\]

\[
\Rightarrow \epsilon_{av}(\pi) \geq \gamma^{n_v - 1} \lim_{N \to \infty} \frac{1}{N + 1} \frac{N}{n_v} = \gamma^{n_v - 1} \frac{1}{n_v} \Rightarrow \gamma^{n_v} \geq \left( \frac{\gamma^{n_v - 1}}{n_v} \right) \Rightarrow \gamma \geq 1/n_v,
\]

which is a contradiction with the assumption that \( \gamma < 1/n_v \). Then a necessary condition for \( \epsilon_{av}(\pi) = \gamma^{n_v} \) with some \( \gamma < 1/n_v \) is the existence of a Hamiltonian cycle.

Remark. Following the reasoning of the above proof it is possible to improve the upper bound in (7) as \( R_{av}(v_0) \leq \lambda \left( 1 - \gamma^p / n_v \right) / (1 - \gamma) \) for small values of \( \gamma \), where \( p \) is the length of the longest simple cycle of the graph.

Theorem 9. The infinite horizon value decision problem is NP-hard for \( \gamma < 1/|V| \).

Proof. We reduce the Hamiltonian cycle problem to the infinite horizon optimal path problem. Given a graph \( G = (V, E) \), we fix some \( \lambda \) and \( \gamma < 1/|V| \). We show that \( G \) is Hamiltonian iff \( R_{av}(v_0) \geq \lambda \left( 1 - \gamma^{|V|} \right) / (1 - \gamma) \). If \( G \) has a Hamiltonian cycle \( v_1, v_2, \ldots, v_{|V|}, v_1 \), then the infinite path \( \pi = (v_1, v_2, \ldots, v_{|V|}, v_1) \) has reward \( r_{av}(\pi) = \lambda \left( 1 - \gamma^{|V|} \right) / (1 - \gamma) \), for any choice of \( \gamma \). For the other direction, applying Lemma 8 with \( \gamma < 1/|V| \) implies that \( G \) is Hamiltonian.

Non-discounted rewards (\( \gamma = 1 \)) and node-invariant function \( \lambda \). Contrary to the finite-horizon non-discounted case, the infinite-horizon optimal path and value problems for \( \gamma = 1 \) can be solved in polynomial time. To see this, note that the reward expression \( r_{av}(\pi) \) can be written as \( r_{av}(\pi) = \lambda \sum_{v \in V} y(v, \pi, N) \), where \( y(v, \pi, N) \) is defined as \( y(v, \pi, N) = 1 + \max U \), for \( U = \{ j \in [N] | j \leq N, \pi[j] = v \} \cup \{-1\} \). Then, we can bound the reward by

\[
W_{av}(\pi) \leq \lim_{N \to \infty} \frac{\lambda \sum_{i=1}^{|V(\pi)|} (N + 1 - i + 1)}{N + 1} = \lim_{N \to \infty} \frac{\lambda |V(\pi)| (2N - |V(\pi)| + 3)}{2(N + 1)} = \lambda |\text{Inf}(\pi)|,
\]

where \( \text{Inf}(\pi) \) is the set of nodes visited in the path \( \pi \) infinitely often. This indicates that the maximum reward is bounded by \( \lambda \) times the maximal size of a reachable SCC in the graph \( G \). This upper bound is also achievable: we can construct an optimal path by finding a maximal SCC reachable from the initial node and a (not necessarily simple) cycle \( v_1 \ldots v_n v_1 \) that visits all the nodes in this SCC. Then, a subset of optimal paths contains paths of the form \( \pi_0 \cdot (v_1 \ldots v_n)^\omega \), where \( \pi_0 \) is any finite path that reaches \( v_1 \). This procedure can be done with a computational time polynomial in the size of \( G \).
Example 10. Consider the graph in Figure 1. The optimal reward for the infinite-horizon non-discounted case is $4\lambda$, achievable by the path $\pi = (abcd)\omega$. Another path which is not ultimately periodic but achieves the same reward $\pi' = (abc)(ad)(abc)^2(ad)^2 \ldots (abc)^n(ad)^n \ldots$.

Note that for the case $\gamma = 1$ there always exists an ultimately periodic optimal path, such a path is generated by a finite-memory strategy.

4.2 Approximate Computation of $R_{av}(v)$

In the previous section we discussed how to solve the infinite-horizon value and path computation problems for the non-discounted case. Now we show how the infinite-horizon path and value computation problems for $\gamma < 1$ can be effectively approximated. We first define functions that over and underapproximate $C_{av}(v)$ (thus also $R_{av}(v)$) and establish bounds on the error of these approximations. Given an integer $K \in \mathbb{N}$, approximately optimal paths and an associated interval approximating $C_{av}(v)$ can be computed using a finite augmented graph $\tilde{G}_K$ based on the augmented graph $G$ of Section 3. Intuitively, $\tilde{G}_K$ is obtained from $G$ by pruning nodes that have a component greater than $K$ in their augmentation. By increasing the value of $K$, the approximation error can be made arbitrarily small.

We describe the approximation algorithm for node-invariant $\lambda$ and $\gamma$. The results generalize trivially to the case when $\lambda$ and $\gamma$ are not node-invariant by choosing $K$ large enough to satisfy the condition that bounds the approximation error for each $\lambda(v)$ and $\gamma(v)$.

Approximate cost functions. Consider the following functions from $V^* \times \mathbb{N}$ to $\mathbb{R}_{\geq 0}$:

$$\tilde{c}_{\text{sum}}^{(N)}(\pi, K) := \sum_{t=0}^{N} \gamma^{\min\{K, \text{Last}_\pi(t, \pi[t])\}}$$ and

$$L_{\text{sum}}^{(N)}(\pi, K) := \sum_{t=0}^{N} \gamma^{\text{Last}_\pi(t, \pi[t])}(\text{Last}_\pi(t, \pi[t]) \leq K).$$

Informally, for $\tilde{c}_{\text{sum}}^{(N)}(\pi, K)$, if the last visit to node $\pi[t]$ occurred more than $K$ time units before time $t$, the cost is $\gamma^K$, rather than the original smaller amount $\gamma^{\text{Last}_\pi(t, \pi[t])}$. For $L_{\text{sum}}^{(N)}(\pi, K)$, if the last visit to $\pi[t]$ occurred more than $K$ time steps before time $t$, then the cost is 0. For both, if the last visit to the node $\pi[t]$ occurred less than or equal to $K$ steps before, we pay the actual cost $\gamma^{\text{Last}_\pi(t, \pi[t])}$. The above definition implies that $L_{\text{sum}}^{(N)}(\pi, K) \leq \tilde{c}_{\text{sum}}^{(N)}(\pi, K) \leq \bar{c}_{\text{sum}}^{(N)}(\pi, K)$ for every $\pi$. Then we have $\underline{C}(v_0, K) \leq C_{av}(v_0) \leq \bar{C}(v_0, K)$, where we define

$$\underline{C}(v_0, K) := \inf_{\pi, \pi[0]=v_0} \limsup_{N \to \infty} \frac{\tilde{c}_{\text{sum}}^{(N)}(\pi, K)}{N+1}$$ and

$$\bar{C}(v_0, K) := \inf_{\pi, \pi[0]=v_0} \limsup_{N \to \infty} \frac{L_{\text{sum}}^{(N)}(\pi, K)}{N+1}.$$
Therefore $C_{av}(v_0)$ belongs to the interval $[\bar{C}(v_0, K), \underline{C}(v_0, K)] \subset [0, \gamma^K]$ and the length of the interval is at most $\gamma^K$. In order to guarantee the total error of $\epsilon > 0$ for the actual reward $R_{av}(v_0)$, we can select $K \in \mathbb{N}$ according to $\frac{1}{\gamma} K < \epsilon \Rightarrow K \geq \ln \left[ \frac{\epsilon (1 - \gamma)}{\lambda} \right] / \ln \gamma$.

The value $C_{av}(v_0)$ can be computed up to the desired degree of accuracy by computing either $\bar{C}(v_0, K)$ or $\underline{C}(v_0, K)$. Next, we give the procedure for computing $\bar{C}(v_0, K)$ (the procedure for $\underline{C}(v_0, K)$ is similar). It utilizes a finite augmented weighted graph $\tilde{G}_K$.

**Truncated augmented weighted graph $\tilde{G}_K$.** Recall the infinite augmented weighted graph $\tilde{G}$ from Section 3. We define a truncated version $\tilde{G}_K = (\tilde{V}_K, \tilde{E}_K)$ of $\tilde{G}$ where we only keep track of last$_\pi(t, v)$ values less than or equal to $K$. For $\tilde{G}_K$ we define two weight functions $\tilde{\pi}$ and $\tilde{w}$, for $\tilde{c}_{\text{sum}}$ and $\tilde{c}_{\text{sum}}^{(N)}$ respectively.

- The set of nodes $\tilde{V}_K$ is $V \times \mathbb{Z}[0, K]|V|$.
- For a node $\tilde{v} = (v, b_1, b_2, \ldots, b_{|V|}) \in \tilde{V}$,
  - the weight $\tilde{\pi}(\tilde{v})$ is $\gamma^{b_u}$ if $b_u > 0$ and $\gamma^K$ otherwise.
  - the weight $\tilde{w}(\tilde{v})$ is $\gamma^{b_u}$ if $b_u > 0$ and 0 otherwise.
- The set of edges $\tilde{E}_K$ consists of edges $(v, b_1, b_2, \ldots, b_{|V|}) \rightarrow (v', b'_1, b'_2, \ldots, b'_{|V|})$ such that $(v \rightarrow v') \in E$, $b'_u = 1$, and for $u \neq v$:
  - $b'_u = b_u + 1$ if $b_u > 0$ and $b_u + 1 \leq K$,
  - $b'_u = 0$ if $b_u = 0$ or $b_u + 1 > K$.

Thus, in $\tilde{G}_K$ we specify two different weights for path $\pi$ at time step $t$ in the case when the previous visit to $n[t]$ in $\pi$ was more than $K$ time steps ago.

Similarly to the infinite augmented graph we have

- $\tilde{c}_{\text{sum}}^{(N)}(\pi, K)$ is the sum of weights assigned by $\tilde{\pi}$ to the first $(N + 1)$ nodes of $\tilde{\pi}$.
- $\tilde{c}_{\text{sum}}^{(N)}(\pi, K)$ is the sum of weights assigned by $\tilde{w}$ to the first $(N + 1)$ nodes of $\tilde{\pi}$.

It is easy to see that $\bar{C}(v_0, K)$ is the least possible limit-average cost with respect to $\tilde{\pi}$ in $\tilde{G}_K$ starting from the node $\tilde{v}_0 = (v_0, 1, 1, \ldots, 1)$. The same holds for $\underline{C}(v_0, K)$ with $\tilde{w}$. Below we show how to compute $\bar{C}(v_0, K)$. The case $\underline{C}(v_0, K)$ is analogous, and thus omitted.

**Algorithm for computing $\bar{C}(v_0, K)$.** We now describe a method to compute $\bar{C}(v_0, K)$ as the least possible limit-average cost in $\tilde{G}_K$ with respect to $\tilde{\pi}$. It is well-known that this can be reduced to the computation of the minimum cycle mean in the weighted graph [26], which in turn can be done using the algorithm from [16] that we now describe.

As before, first we assume that $\tilde{G}_K$ is strongly connected. For every $\tilde{v} \in \tilde{V}_K$, and every $n \in \mathbb{Z}_+$, we define $W_n(\tilde{v})$ as the minimum weight of a path of length $n$ from $\tilde{v}_0$ to $\tilde{v}$; if no such path exists, then $W_n(\tilde{v}) = \infty$. The values $W_n(\tilde{v})$ can be computed by the recurrence

$$W_n(\tilde{v}) = \min_{(\tilde{u}, b) \in \tilde{E}_K} \{W_{n-1}(\tilde{u}) + \tilde{w}(\tilde{v})\}, \quad n = 1, 2, \ldots, |\tilde{V}_K|,$$

with the initial conditions $W_0(\tilde{v}_0) = 0$ and $W_0(\tilde{v}) = \infty$ for any $\tilde{v} \neq \tilde{v}_0$. Then, we can compute

$$\bar{C}(v_0, K) = \min_{\tilde{v} \in \tilde{V}_K} \max_{n \in \mathbb{Z} \cup \{0, |\tilde{V}_K| - 1\}} \left[ \frac{W_n(\tilde{v})}{|\tilde{V}_K| - n} - W_n(\tilde{v}) \right].$$

A cycle $\tilde{\pi}$ with the computed minimum mean can be extracted by fixing the node $\tilde{v}$ which achieves the minimum in the above value and the respective path length $n$ and finding a

---

1 Since $R_{av}(v_0)$ is upper bounded by $\lambda/(1 - \gamma)$, we assume that the required accuracy $\epsilon$ is less than this upper bound.
minimum-weight path from $\tilde{v}_0$ to $\tilde{v}$ and a cycle of length $|\tilde{V}_K| - n$ within this path. Thus, the path $\tilde{\pi}$ in $\tilde{G}_K$ obtained by repeating $\tilde{\rho}$ infinitely often realizes this value. A path $\pi$ from $v_0$ in $G$ with $c_{av}(\pi) = \overline{C}(v_0, K)$ is obtained from $\tilde{\pi}$ by projection on $V$.

In the general case, when $\tilde{G}_K$ is not strongly connected we have to consider each of its SCCs reachable from $\tilde{v}_0$, and determine the one with the least minimum cycle mean.

For each SCC with $m$ edges and $n$ nodes the computation of the quantities $W$ requires $O(n \cdot m)$ operations. The computation of the minimum cycle mean for this component requires $O(n^2)$ further operations. Since $n \leq m$ because of the strong connectivity, the overall computation time for the SCC is $O(n \cdot m)$. Finally, the SCCs of $\tilde{G}_K$ can be computed in time $O(|\tilde{V}_K| + |\tilde{E}_K|)$ [21]. This gives us the following result.

\begin{itemize}
  \item [\textbf{Lemma 11.}] The value $\overline{C}(v_0, K)$ and a path $\pi$ with limit average cost $c_{av}(\pi) = \overline{C}(v_0, K)$ can be computed in time polynomial in the size of $\tilde{G}_K$.
\end{itemize}

The same result can be established for the under approximation $\underline{C}(v_0, K)$.

\begin{itemize}
  \item [\textbf{Remark.}] The number of nodes of $\tilde{G}_K$ is $|\tilde{V}_K| = |V| \times (K + 1)^{|V|}$. For the approximation procedure described above it suffices to augment the graph with the information about which nodes were visited in the last $K$ steps and in what order. Thus, we can alternatively consider a graph with $|V| \times |V|^K$ nodes in the case when the computed $K$ is smaller than $|V|$.
\end{itemize}

\begin{itemize}
  \item [\textbf{Example 12.}] Figure 2 shows the SCC of the augmented graph $\tilde{G}_K$ reachable from initial node $(a, 1, 1, 1, 1)$ for the graph given in Figure 1 and $K = 5$. The nodes in the SCC are denoted by shorthands in the picture, e.g., $a_1 = (a, 3, 2, 1, 4)$. The labels on the nodes correspond to the values of the weight functions $\overline{w}$ and $\underline{w}$. As we can see, $\tilde{G}_5$ already contains simple cycles $(a_2b_2c_2a_1b_1a_2d_2a_3), (a_3b_3c_2a_1b_1a_2d_2a_3), (a_3b_3c_2a_1d_1a_3)$, which correspond to the optimal paths presented in Example 3. The outcome of the minimum cycle mean for $\tilde{G}_5$ will be the same for the two sets of weights only for the first and third interval for $\gamma$ determined in Example 3, but will be different for the second case in which the term $\gamma^8$ is replaced respectively by $\gamma^5$ and $0$ for the upper and lower bounds.
\end{itemize}

Theorem 13, a consequence of Lemma 11, states the approximate computation result.

\begin{itemize}
  \item [\textbf{Theorem 13.}] Given a graph $G = (V, E)$, node $v_0 \in V$, rational values $\lambda$ and $0 < \gamma < 1$, and error bound $\epsilon > 0$, we can compute in time polynomial in $|V|(K + 1)^{|V|}$ for $K =$...
13:14 The Robot Routing Problem for Collecting Aggregate Stochastic Rewards

\[ \ln \left( \frac{1 - e^{-\gamma}}{\lambda} \right) / \ln \gamma \text{ (i.e., exponential in } |V| \text{), infinite paths } \pi_{\text{under}} \text{ and } \pi_{\text{over}} \text{ and values } r_{\text{av}}(\pi_{\text{under}}) \text{ and } r_{\text{av}}(\pi_{\text{over}}) \text{ such that } r_{\text{av}}(\pi_{\text{under}}) \leq R_{\text{av}}(v_0) \leq r_{\text{av}}(\pi_{\text{over}}) \text{ and } r_{\text{av}}(\pi_{\text{over}}) - r_{\text{av}}(\pi_{\text{under}}) \leq \epsilon. \]

4.3 Approximation via Bounded Memory

The algorithm presented earlier is based on an augmentation of the graph with a specific structure updated deterministically and whose size depends on the desired quality of approximation. Furthermore, in this graph there exists a memoryless strategy with approximately optimal reward. We show that this allows us to quantify how far from the optimal reward value is a strategy that is optimal among the ones with bounded memory of fixed size.

First, we give the definition of memory structures. A memory structure \( M = (M, m_0, \delta) \) for a graph \( G = (V, E) \) consists of a set \( M \), initial memory \( m_0 \in M \) and a memory update function \( \delta : M \times V \rightarrow M \). The memory update function can be extended to \( \delta^* : V^* \rightarrow M \) by defining \( \delta^*(e) = m_0 \) and \( \delta^*(\pi \cdot v) = \delta(\delta^*(\pi), v) \). A memory structure \( M \) together with a function \( \tau : V \times M \rightarrow V \) such that \( (v, \tau(v, m_0)) \in E \) for all \( v \in V \) and \( m \in M \), and an initial node \( v_0 \in V \) define a strategy \( \sigma : V^* \rightarrow V \) where \( \sigma(\epsilon) = v_0 \) and \( \sigma(\pi \cdot v) = \tau(v, \delta^*(\pi)) \). In this case we say that the strategy \( \sigma \) has memory \( M \). Given a bound \( B \in \mathbb{N} \) on memory size we define the finite graph \( G \times B = (V^{G \times B}, E^{G \times B}) \), where \( V^{G \times B} = V \times [1, B] \) and \( E^{G \times B} = \{(v, i), (v', j)\} \in (V \times [1, B]) \times (V \times [1, B]) \mid (v, v') \in E\). Memoryless strategies in this graph precisely correspond to strategies that have memory of size \( B \). More precisely, for each strategy \( \sigma \) in \( G = (V, E) \) that has memory \( M = (M, m_0, \delta) \) there exists a memoryless strategy \( \sigma_M \) in \( G \times |M| \) such that the projection of \( \text{outcome}(v_0, m_0, \sigma_M) \) on \( V \) is \( \text{outcome}(v_0, \sigma) \). Conversely, for each memoryless strategy \( \sigma_M \) in \( G \times B \) there exist a memory structure \( M = (M, m_0, \delta) \) with \( |M| = B \) and a strategy \( \sigma \) with memory \( M \) in \( G \) such that the projection of \( \text{outcome}(v_0, m_0, \sigma_M) \) on \( V \) is \( \text{outcome}(v_0, \sigma) \).

Example 14. Recall that in Example 3 we established that an optimal path for \( \gamma = 0.26 \) is the path \( abcabead \). In the graph \( G \times 3 \) there exists a simple cycle corresponding to this path, namely \( (a, 1)(b, 1)(c, 2)(a, 2)(b, 2)(c, 3)(a, 3)(d, 1)(a, 1) \). Thus, the optimal path \( abcabead \) corresponds to a strategy with memory size of 3. An optimal strategy among those with memory of a given size \( B \) can be computed by inspecting the memoryless strategies in \( G \times B \) and selecting the one with maximal reward (these strategies are finitely but exponentially many).

A strategy returned by the approximation algorithm presented earlier uses a memory structure of size \( (K + 1)^{|V|} \). If \( (K + 1)^{|V|} \leq B \), then this strategy is isomorphic to some memoryless strategy \( \sigma \) in \( G \times B \). Since the reward achieved by the optimal memoryless strategy in \( G \times B \) is at least that of \( \sigma \), its value is at most \( \frac{1}{1 - \gamma} R_{\text{av}}(v_0) \) away from \( R_{\text{av}}(v_0) \). Now, \( (K + 1)^{|V|} \leq B \) if \( K \leq \left[ \frac{\ln B}{\ln |V|^{|V|}} \right] - 1 \). Thus, under a memory size \( B \), we are guaranteed to find a strategy which has reward that at most \( \frac{1}{1 - \gamma} \left[ \frac{\ln B}{\ln |V|^{|V|}} \right]^{-1} \) away from the optimal.

Next we sketch an algorithm for computing optimal \( B \)-memory bounded strategies. As mentioned previously, we can restrict our attention to memoryless strategies in \( G \times B \). Memoryless strategies in this graph lead to lasso shaped infinite paths, with an initial non-cyclic path followed by a simple cycle which is repeated infinitely often. This means that we can restrict our attention to paths of length \( |V| \cdot B \) which complete the lasso. Now, we apply this length bound to restrict our attention to the finite portion of the augmented graph \( G \) from Section 3 that corresponds to path lengths at most \( |V| \cdot B \). This finite subgraph contains nodes \( V \times \mathbb{N}[1, |V| \cdot B]^{|V|} \). The dynamic programming algorithm is polynomial time on this graph, hence we get a running time which is polynomial in \( |V| \cdot (|V| \cdot B)^{|V|} \).
Theorem 15. Given a graph $G = (V, E)$, a node $v_0 \in V$, rational values $\lambda$ and $0 < \gamma < 1$, and a memory bound $B > 1$, we can compute a $B$-memory optimal strategy $\sigma$ with reward $r_{av}$ at most $\frac{\lambda}{1 - \gamma} \left( \frac{|V|}{\ln B} \right)^{-1}$ away from the optimal $R_{av}(v_0)$ in time polynomial in $|V|(|V|+1) \cdot B|V|$.

5 Conclusion and Open Problems

We have introduced the robot routing problem, which is a reward collection problem on graphs in which the reward structure combines spatial aspects with dynamic and stochastic reward generation. We have studied the computational complexity of the finite and infinite-horizon versions of the problem, as well as the strategy complexity of the infinite-horizon case. We have shown that optimal strategies need memory in general, and that strategies based on last visitation records do not suffice. However, the important question about whether finite-memory suffices for the infinite-horizon optimal path problem or infinite memory is needed remains open. In case finite-memory suffices, it will be desirable to provide an algorithm for precisely solving the infinite-horizon value problem. So far we have only given methods for approximating the optimal solution.

Another interesting line of future work is the generalization of the model to incorporate timing constraints. More specifically, in this work we have assumed that all the rewards accumulated at a node are instantaneously collected. This assumption is justified by the fact that in many request-serving scenarios the time taken to serve the requests at a given location is negligible compared to the time necessary to travel between the locations. A more precise model, however, would have to incorporate the serving time per node, which would depend on the amount of collected reward. This would imply that the elapsed time becomes a random variable, while in our current model it is deterministic.

Other extensions include the setting where the robot can react to the environment by observing the collected rewards, or observing the accumulated rewards at nodes of the graph, or where there is probabilistic uncertainty in the transitions of the robot in the graph. Finally, the robot routing problem presents new challenges for development of efficient approximation algorithms, which is a major research direction concerning path optimization problems.

References


5 Patricia Bouyer, Piotr Hofman, Nicolas Markey, Mickael Randour, and Martin Zimmermann. Bounding average-energy games. In Foundations of Software Science and Computa-
The Robot Routing Problem for Collecting Aggregate Stochastic Rewards


Coverability Synthesis in Parametric Petri Nets

Nicolas David\textsuperscript{1}, Claude Jard\textsuperscript{2}, Didier Lime\textsuperscript{3}, and Olivier H. Roux\textsuperscript{4}

\textsuperscript{1} University of Nantes, LS2N, Nantes, France
nicolas.david1@univ-nantes.fr
\textsuperscript{2} University of Nantes, LS2N, Nantes, France
claude.jard@univ-nantes.fr
\textsuperscript{3} École Centrale de Nantes, LS2N, Nantes, France
didier.lime@ec-nantes.fr
\textsuperscript{4} École Centrale de Nantes, LS2N, Nantes, France
olivier-h.roux@ec-nantes.fr

Abstract
We study Parametric Petri Nets (PPNs), i.e., Petri nets for which some arc weights can be parameters. In that setting, we address a problem of parameter synthesis, which consists in computing the exact set of values for the parameters such that a given marking is coverable in the instantiated net.

Since the emptiness of that solution set is already undecidable for general PPNs, we address a special case where parameters are used only as input weights (preT-PPNs), and consequently for which the solution set is downward-closed. To this end, we invoke a result for the representation of upward closed set from Valk and Jantzen. To use this procedure, we show we need to decide universal coverability, that is decide if some marking is coverable for every possible values of the parameters. We therefore provide a proof of its EXPSPACE-completeness, thus settling the previously open problem of its decidability.

We also propose an adaptation of this reasoning to the case of parameters used only as output weights (postT-PPNs). In this case, the condition to use this procedure can be reduced to the decidability of the existential coverability, that is decide if there exists values of the parameters making a given marking coverable. This problem is known decidable but we provide here a cleaner proof, providing its EXPSPACE-completeness, by reduction to ω-Petri Nets.

1998 ACM Subject Classification F.1.1 Models of Computation

Keywords and phrases Petri net, Parameters, Coverability, Unboundedness, Synthesis

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2017.14

1 Introduction

The introduction of parameters in models aims to improve genericity. It also allows the designer to leave unspecified aspects, such as those related to the modeling of the environment. This increase in modeling power usually results in greater complexity in the analysis and verification of the model. Beyond verification of properties, the use of parameters opens the way to very relevant issues in design, such as the computation of the parameters values ensuring satisfaction of the expected properties. This is the synthesis problem: given a property, compute the exact set of all parameter values such that, instantiated with these values, the system satisfies this property. This notably permits an estimation of the robustness

* This work was partially supported by ANR project PACS (ANR-14-CE28-0002) and Pays de la Loire research project AFSEC.
Coverability Synthesis in PPNs

of a given instance of a model. Indeed, in full knowledge of “good values” for the parameters, we may be able to quantify the distance from a “bad value” providing an idea of how reliable is the system. Thus parameterised systems are of particular interest both in allowing the handling of more realistic classes of models and addressing more realistic verification issues. We therefore address here the case of parameterised concurrent systems modelled as parametric Petri nets.

Related work. The study of parameterised models and more specifically the synthesis has been studied in different parametrics settings. Parameters representing delays in timed systems modeled as timed automata have been particularly studied, but with very few decidability results [1]. Synthesis for such systems is only possible in very particular settings, such as bounded integer parameters computed symbolically in timed automata [15] or integer parameters in timed automata with parameters used only as upper bounds, or only as lower bounds, in timing constraints [4]. We focus here on a different type of parameters which represent discrete values. In Petri nets, this corresponds to parameterising the initial marking [5, 17] and the weights of the arcs in transitions [7]. The latter work deals with two decidability problems induced by the use of parameters: The existential coverability: does there exists an integer valuation v on the set of parameters such that m is coverable in the marked Petri net where parameters are replaced by the value given by v? And the universal coverability: is m is coverable in such a net for every possible valuation v? Those problems are both undecidable in the most general case, and syntactic subclasses restricting the use of parameters have been introduced, for which the different problems are decidable.

Contributions. We focus on computing the exact solution set to the synthesis problem for coverability in parametric Petri nets, i.e., the set of all parameter values such that in the net instantiated with these values, a given marking is coverable.

The emptiness and universality of the solution set being undecidable in general, computing this set can only be done in a restricted setting. We thus focus on the case when parameters are used only as input weights (preT-PPNs) or only as output weights (postT-PPNs). These assumptions give some structure to the solution set: we prove that it is then downward-closed wrt. the usual order on integer vectors for preT-PPNs, and upward-closed for postT-PPNs.

We show how a procedure by Valk and Jantzen from [20] can be used for computing a finite minimal basis of the solution set for postT-PPNs or its complement for preT-PPNs. This requires deciding universal coverability in preT-PPNs and existential coverability in postT-PPNs. The former is an open problem: our main result is it is ExpSPACE-complete, which we prove by considering the more generic property of simultaneous unboundedness studied by Demri in [8]. The latter is known decidable but we provide a cleaner proof and additionally prove its ExpSPACE-completeness. These results interestingly allow us to carry over a Rackoff upper bound into this parametric setting.

Finally, we prove that in what is called distinctT-PPNs in [7], i.e., when the set of parameters appearing as input weights, and the set of parameters appearing as output weights are disjoint, the solution set cannot be represented using any formalism for which the emptiness of the intersection with equality constraints is decidable.

Organization of the Paper. Section 2 gives basic notations and recalls useful mathematical results on orders and Petri nets. Section 3 presents Parametric Petri Nets and recalls some decidability problems. There, we also study the structure of the solution sets for preT- and postT-PPNs and show under which condition Valk and Jantzen’s algorithm can be used.
to construct finite representation of those sets. In section 4 we give our construction for proving the \( \text{ExpSpace}\)-completeness of the universal coverability for preT-PPNs. Section 5 revisits the proof of the decidability of existential coverability in postT-PPNs and proves its \( \text{ExpSpace}\)-completeness. We also discuss the case of distinctT-PPNs. Finally, in Section 6, we conclude and present future work. Due to space contraints, proofs are omitted.

## 2 Background

### 2.1 Notations

We denote by \( \mathbb{Z} \) the set of integers, and by \( \mathbb{N} \) the set of natural numbers. As usual, \( \mathbb{N}_\omega \) is the union \( \mathbb{N} \cup \{ \omega \} \) where for each \( n \in \mathbb{N}, n + \omega = \omega, \omega - n = \omega \) and \( \omega \leq \omega \). Moreover, if \( n \in \mathbb{N}, n < \omega \). Let \( X \) be a finite set. We denote by \( 2^X \) the powerset of \( X \) and \( |X| \) the size of \( X \). If \( X \subseteq \mathbb{N}_\omega \), \( \neg X \) denotes its complement in \( \mathbb{N}_\omega \). Given a finite set \( X \), \( S_X \) denotes the symmetric group on \( X \) (i.e. the set of all permutations of elements of \( X \)). Given a set \( X \), we define a linear expression on \( X \) by the following grammar: \( \lambda ::= k | k \cdot x | \lambda + \lambda \) where \( k \in \mathbb{Z}, x \in X \). We denote by \( L(X) \) the set of linear expressions on \( X \).

Let \( V \subseteq \mathbb{N} \), a \( V \)-valuation for \( X \) is a function from \( X \) to \( V \). We denote by \( V^X \) the set of \( V \)-valuations on \( X \). Considering \( v \in V^X \), we write \( \text{dom}(v) \) the domain of \( v \) (\( X \) in this case) and \( \text{im}(v) \) its image. We refer to \( \mathbb{N}_\omega \)-valuations as extended valuations and to \( \mathbb{N} \)-valuations simply as valuations. The set \( V^\emptyset \) is reduced to a singleton \( \{ \emptyset_v \} \) where \( \emptyset_v \) is the empty function. If \( X \) is finite, considering some arbitrary order on \( X \), an (extended) valuation can be seen as a vector of size \( |X| \). For any subset \( X' \subseteq X \) and valuation \( v \in V^X \), we define the restriction \( v_{|X'} \) of \( v \) to \( X' \) as the unique \( V \)-valuation on \( X' \) such that \( v_{|X'}(x) = v(x) \) for all \( x \in X' \). We extend this notation to sets of valuations: given \( Y \subseteq V^X \), \( Y_{|X'} \) denotes its projection on \( X' \) that is to say \( Y_{|X'} = \{ v_{|X'} \mid v \in Y \} \). Given a value \( a \in \mathbb{N}_\omega \), we denote as \( a \) the uniform (extended) valuation that maps every element of \( X \) to \( a \). Given an extended valuation \( v \), we write \( \omega(v) \) for the subset of \( X \) such that \( x \in \omega(v) \) iff \( v(x) = \omega \). We write \( \mathbb{N}(v) \) for the subset of \( X \) such that \( x \in \mathbb{N}(v) \) iff \( v(x) \in \mathbb{N} \).

Given a linear expression \( \lambda \) on \( X \) and an extended valuation \( v \) on \( X' \subseteq X \), \( v(\lambda) \) is the linear expression obtained when substituting each element \( x \) in \( X' \) from \( \lambda \), by the corresponding value \( v(x) \). If \( X' = X \) we obtain an element of \( \mathbb{N}_\omega \).

Given a set \( R \), finite sets \( S, A, B \) such that \( S = A \cup B \) and \( A \cap B = \emptyset \), and functions \( f \in R^A \) and \( g \in R^B \), we write \( f \cup g \) the function defined by \( (f \cup g)|_A = f \) and \( (f \cup g)|_B = g \). We call \( f \cup g \) the union of \( f \) and \( g \). Note that given \( x \) in \( A \), \( y = f(x) \) is called the image of \( x \) and when there is no ambiguity (i.e. when \( f \) is injective), \( x \) is called the fiber of \( y \) by \( f \).

Finally, let \( A \) be an alphabet and \( A^* \) be the free monoid over \( A \). Let \( w \in A^* \) be a word. We write \( |w| \) the length of \( w \). Given \( a \in A \), \( |w|_a \) is the number of occurrences of \( a \) in \( w \). We define \( \epsilon \) as the identity element of \( A^* \). We write \( t \sqsubseteq s \) when \( t \) is a prefix of \( s \). We denote by \( \text{Pref}(L) \) the prefix closure of a langage \( L \), i.e. \( \text{Pref}(L) = \bigcup_{s \in L} \{ t \mid t \sqsubseteq s \} \).

### 2.2 Order

A quasi order (\( \sqsubseteq \) for short) \( \sqsubseteq \) on some set \( S \) is a reflexive and transitive binary relation on \( S \). The pair \( (S, \sqsubseteq) \) is then called a quasordered set. For \( x, y \in S \) and given a \( \sqsubseteq \) on \( S \), \( x \) and \( y \) are said comparable if either \( x \sqsubseteq y \) or \( y \sqsubseteq x \). A relation \( < \) is a strict order on a set \( S \) if it is irrelexive and transitive (which implies asymmetry).
Given any quasi order $\preceq$ on a set $S$ we can define: (i) a strict order $<$ given by $x < y$ iff $x \preceq y \wedge \neg (y \preceq x)$, (ii) an equivalence relation $\sim$ given by $x \sim y$ iff $x \preceq y \wedge y \preceq x$, (iii) its dual quasi order $\succeq$ given by $y \succeq x$ if $x \preceq y$. A well-quasi-ordering (wqo for short) is a qo $\preceq$ on a set $S$ such that, for any infinite sequence $s = x_0, x_1, x_2, \ldots$ in $S$, there exists indexes $i < j$ with $x_i \preceq x_j$. Now consider $\leq$, the qo on $\mathbb{N}^k$ component-wise. Formally, for every $x, y \in \mathbb{N}^k$, we write that $x \leq y$ iff for every component $i$ of $x$ and $y$, $x(i) \leq y(i)$. Dickson’s Lemma [9] states that $(\mathbb{N}^k, \leq)$ is a well-quasi-ordered set. Let us also recall the following lemma:

\begin{lemma} [9] \end{lemma}

Let $p_0, p_1, \ldots, p_n, \ldots$ be an infinite sequence of elements of $(\mathbb{N}^k)^{\omega}$. Then, there is an infinite sequence $p_{i_1}, p_{i_2}, \ldots, p_{i_n}, \ldots$ such that $p_{i_1} \leq p_{i_2} \leq \ldots \leq p_{i_n} \leq \ldots$ (with $i_1 < i_2 < \cdots < i_n < \ldots$).

### 2.3 Downward and Upward closed sets

We reuse definitions and concepts from [10, 11] which are summed up in [12]. An upward closed set of the well quasi ordered set $(\mathbb{N}^k, \preceq)$ is a subset $U$ of $\mathbb{N}^k$ such that if $x \in U$, $y \in \mathbb{N}^k$ and $x \preceq y$ then $y \in U$. The upward closure of a vector $u$, written $\uparrow u$ is the set \{ $m \in \mathbb{N}^k$ | $u \preceq m$ \}. Given a set $U$, we write $\uparrow U$ for the upward closure of $U$, defined as $\uparrow U = \bigcup_{u \in U} \uparrow u$. This implies that $\uparrow U$ is the least upward closed set in which $U$ is included.

Any upward closed set $U$ can be represented by a finite set $F$, called basis, such that $U = \uparrow F$. The minimal elements of $F$ still form a basis of $U$ independently of $F$. This basis is minimal for inclusion among all bases and is thus called the minimal upward basis of $F$.

A downward closed set of the well quasi ordered set $(\mathbb{N}^k, \preceq)$ is a subset $D$ of $\mathbb{N}^k$ such that if $x \in D$, $y \in \mathbb{N}^k$ and $y \preceq x$ then $y \in D$. The downward closure of a vector $d$, written $\downarrow d$ is the set \{ $m \in \mathbb{N}^k$ | $m \preceq d$ \}. Given a set $D$, we write $\downarrow D$ the downward closure of $D$, defined as $\downarrow D = \bigcup_{d \in D} \downarrow d$. This implies that $\downarrow D$ is the least downward closed set in which $D$ is included. Moreover, the downward closure of a finite set is finite. To symbolically represent downward closed sets, we use the extension $\mathbb{N}^k_\subseteq$. The definitions remain otherwise the same. If $D$ is a downward closed set, we can write $D = \mathbb{N}^k \cap \downarrow D$ where $F$ is a finite set of $\mathbb{N}^k_\subseteq$. We call $F$ a downward basis of $D$. The maximal elements of $F$ still form a basis of $D$ independently of $F$. This basis is minimal for the inclusion among all bases and is thus called the minimal downward basis of $D$.

We also recall results from [3]: the union and the intersection of two upward (resp. downward) closed sets is an upward (resp. downward) closed set. The complement of an upward closed set is a downward closed set and vice-versa. Given the basis of an upward closed set, it is possible to compute the basis of its complement using for instance the procedure suggested in Example 5 of [14], and vice versa by adapting this procedure.

Finally, Valk and Jantzen proposed in [20] a necessary and sufficient condition, recalled in Lemma 2, to ensure that a finite basis of an upward closed set is effectively computable.

\begin{lemma} [20] \end{lemma}

Given an upward closed set $U \subseteq \mathbb{N}^k$, a finite basis of $U$ is effectively computable iff for each $v \in \mathbb{N}^k$, the emptiness of $\downarrow v \cap U$ is decidable, which is also equivalent to ask whether for all element $v \in \mathbb{N}^k_\subseteq$, it is decidable to answer whether $\downarrow v \cap \mathbb{N}^k_\subseteq \subseteq \neg U$.

---

1 The existence of such increasing subsequences can also be used as a definition for wqo, which leads to an equivalent notion. Note that a proof can also be found in [16].
2.4 Petri Nets

Definition 3 (Marked Petri Net). A Petri net $N$ is a tuple $(P,T, Pre, Post)$ such that $P$ is a finite set of places of $S$, $T$ is a finite set of transitions of $S$, $Pre$ and $Post$ are functions from $P \times T$ to $\mathbb{N}$. A marking of $N$ is an $\mathbb{N}$-valuation on $P$. A marked Petri Net (PN) is a pair $\mathcal{N} = (N,m_0)$ where $N$ is a Petri net and $m_0$ the initial marking of $N$.

Given a transition $t \in T$, we define $Pre(t)$ as the univariate function on $P$ at the point $t$ which associates to each $p \in P$ the weight $Pre(p,t)$. We define $Post(t)$ in a similar way. A transition $t \in T$ is said enabled by a marking $m$ when $m \geq Pre(t)$.

Definition 4 (PN Semantics). The semantics of a PN is a transition system $\mathcal{S}_T = (Q,q_0,\rightarrow)$ where, $Q = NP$, $q_0 = m_0$, $\rightarrow \subseteq Q \times T \times Q$ such that for all $t \in T$, $m \xrightarrow{t} m' \iff m \geq Pre(t)$ and $m' = m - Pre(t) + Post(t)$.

This relation can be extended to sequences of transitions as follows: (i) $m \xrightarrow{t_1 \ldots \rightarrow t_n} m'$ if $m = m'$ (ii) $m \xrightarrow{w} m'$ if $\exists m'', m \xrightarrow{w} m'' \land m'' \xrightarrow{t} m'$ where $w \in T^*$ and $t \in T$. We write $\xrightarrow{\rightarrow}$ the reflexive transitive closure of $\rightarrow$, i.e., $m \xrightarrow{\rightarrow} m'$ when there exists $w \in T^*$ such that $m \xrightarrow{w} m'$.

Definition 5 (Reachability). Let $S = (\mathcal{N},m_0)$, where $\mathcal{N} = (P,T,Pre,Post)$, a marking $m$ of $NP$ is reachable in $S$ iff $m_0 \xrightarrow{\rightarrow} m$.

The reachability set $RS(S)$ of $S$ is the set of all reachable markings of $S$.

Definition 6 (Coverability). Let $S = (\mathcal{N},m_0)$, where $\mathcal{N} = (P,T,Pre,Post)$, and $m$ a marking of $NP$, $m$ is coverable in $S$ iff $\exists m' \in RS(S), m' \geq m$.

The coverability set $CS(S)$ of $S$ is the set of markings coverable in $S$. Coverability is decidable in marked Petri nets [16]. The coverability set is an over approximation of the reachability set in the sense that $CS(S) \supseteq RS(S)$. Given a marked PN $S$, and a marking $m$, we denote by $cov(S,m) \in \{True, False\}$ the coverability of $m$ in $S$. In Petri nets, coverability allows to verify safety properties. We recall that the coverability set of a marked Petri net is computable in the sense that its minimal downward basis is computable (see, e.g., [12]).

3 Monotonicity in Parametric Petri Nets

3.1 Parametric Petri Nets and Parametric Problems

Following [7], we recall the definitions related to marked Parametric Petri Nets (PPNs). We omit the case of parametric initial markings which is a subcase of parametric output weights.

Definition 7 (Parametric Petri Net). A marked parametric Petri Net (PPN) is a pair $\mathcal{S} = (\mathcal{N},m_0)$ where $\mathcal{N} = (P,T,Pre,Post,\mathbb{P})$ such that $P$ is a finite set of places of $\mathcal{N}$, $T$ is a finite set of transitions of $\mathcal{N}$, $\mathbb{P}$ is a finite set of parameters of $\mathcal{N}$, Pre and Post are functions from $P \times T \times \mathbb{P}$ to $\mathbb{N} \cup \mathbb{P}$, $m_0$ is the initial marking of $\mathcal{N}$ belonging to $NP$.

We define the parametric transitions of $\mathcal{S}$, $\Theta \subseteq T$ as the set of transitions with at least one parameter on an input or output arc: $\Theta = \{t \in T \mid \exists p \in P \text{ s.t. } Pre(p,t) \in \mathbb{P} \lor Post(p,t) \in \mathbb{P}\}$. We refer to $T \setminus \Theta$ as the set of plain transitions in echo to the notations of [13].

Considering an arbitrary ordering on places, parametric markings can be represented as vectors of linear combinations on the set of parameters i.e. from $\mathcal{L}(\mathbb{P})^{\mid P \mid}$. Similarly, Pre and Post can be seen as matrices of $(\mathbb{N} \cup \mathbb{P})^{\mid P \mid \times \mid T \mid}$.
Given a marked PPN $S = (\mathcal{N}, m_0)$, where $\mathcal{N} = (P, T, Pre, Post, \mathcal{P})$, for any $\mathbb{N}$-valuation $v$ on a subset $X$ of $\mathcal{P}$, we define the $v$-instance of $S$ as the marked PPN $v(S) = (v(\mathcal{N}), m_0)$ where $v(\mathcal{N}) = (P, T, v(Pre), v(Post), \mathcal{P}, X)$. By $v(Pre)$ and $v(Post)$ we denote the function-s/matrices obtained by replacing in their entries each parameter $\lambda$ in $\text{dom}(v)$ by $v(\lambda)$. If $X = \mathcal{P}$, $v(\mathcal{N})$ and $v(S)$ are respectively a Petri net and a marked Petri net. We also recall subclasses introduced in [7]. Given a PPN, $\mathcal{N} = (P, T, Pre, Post, \mathcal{P})$, if $Pre \in P \times T \rightarrow \mathbb{N}$, we call it a postT-PPN, whereas if $Post \in P \times T \rightarrow \mathbb{N}$, we call it a preT-PPN.

Given a PPN $S$, and a marking $m$, we define two basic parametric decision problems: Does there exist a valuation $v$ such that $m$ is coverable in $v(S)$ (Existential coverability) ? Is $m$ coverable in $v(S)$ for all valuations $v$ (Universal coverability) ? Those problems were partly studied in [7]. In particular both Existential coverability and Universal coverability are proved to be undecidable for the generic class of PPNs. In this paper, we are interested in a more general question:

Definition 8 (coverability-Synthesis problem). Compute all the valuations $v$, such that $\text{cov}(v(SP), m)$ is true.

We call this set of valuations the coverability synthesis set of a marked PPN $S$ and a marking $m$, denoted by $\text{CV}(S, m)$. We also call it the solution set to the synthesis problem.

Remark. From any PN $S$, we can build a PPN $S'$ by adding an unused parameter. Then checking existential or universal coverability on $S'$ is equivalent to checking coverability on $S$. Those parametric problems are therefore EXPSPACE-hard. The same reasoning applies for other properties such as (simultaneous) unboundedness.

3.2 Special Structure of the Coverability Synthesis Set for PreT-PPNs and PostT-PPNs

When we restrict the use of parameters to input arcs, we ensure that any marking coverable in a $v$-instance remains coverable for any $v'$-instance such that $v' \leq v$. Intuitively, decreasing the valuation leads to a more permissive firing condition. Symmetrically, when we restrict the use of parameters to output arcs, we ensure that any marking coverable in a $v$-instance, remains coverable for any $v'$-instance such that $v \leq v'$. Intuitively, firing the same parametric transition while increasing the valuation leads to greater (and thus more permissive) markings. Those results are formalized in Lemma 9.

Lemma 9. Let $S_{pre}$ and $S_{post}$ be respectively a marked preT-PPN and postT-PPN of initial marking $m_0$ and so.

- For every transitions sequence $w$ of $S_{pre}$ and for every valuation $v$, if $m_0 \xrightarrow{w} m$ in $v(S_{pre})$, then for every valuation $v' \leq v$, there exists $m' \geq m$ such that $m_0 \xrightarrow{w} m'$ in $v'(S_{pre})$.

- For every transitions sequence $w$ of $S_{post}$ and for every valuation $v$, if $s_0 \xrightarrow{w} s$ in $v(S_{post})$, then for every valuation $v' \geq v$, there exists $s' \geq s$ such that $s_0 \xrightarrow{w} s'$ in $v'(S_{post})$.

Note that those properties of monotonicity directly provide a notable structure for the solution set of the synthesis for those two subclasses presented in Corollary 10.

Corollary 10. Given $S_{pre}$, $S_{post}$ a marked preT-PPN and a marked postT-PPN respectively and goal markings $m$ and $s$ for each of those nets,

- $\text{CV}(S_{pre}, m)$ is downward closed.

- $\text{CV}(S_{post}, s)$ is upward closed.
3.3 Reduction of Valk and Jantzen Condition for PreT-PPNs and PostT-PPNs

Given a preT-PPN $S$ and a marking $m$, one way to compute $CV(S, m)$ is thus to find its finite minimal basis. A naive enumeration is not possible however since this set may be infinite. In particular, the strategy that consists in testing for universality and, in the negative case, enumerating until a witness of non coverability is found would in general provide only a subset of $CV(S, m)$. In fact, the main difficulty here resides in the fact that the elements of the minimal basis have to be found among the complete lattice induced by $\leq$ on $\mathbb{N}_P^\omega$.

In order to represent a finite basis of a downward closed set of valuations, we need to extend valuations to $\mathbb{N}_\omega$. Given a preT-PPN and an extended valuation $v$, we extend the predicate $cov(v(S), m)$ to extended valuations as follows: $cov(v(S), m) \overset{def}{=} \forall v' \in \mathbb{N}^{\omega(v)}, cov(v'(v|_{\mathbb{N}(v)}(S)), m)$.

Figure 1 presents a preT-PPN with two parameters $a$ and $b$. If we consider the valuation $v$ defined by $v(a) = 1$ and $v(b) = \omega$, $cov(v(S), m)$ is therefore equivalent to the universal coverability of $m$ in $v_{\{a\}}(S)$ where $v_{\{a\}}$ is a valuation defined by $v(a) = 1$, that is to say “can we cover $m$ in $1_{\{a\}}(S)$ for any value of $b$ ?”. Note that this extension of $cov(v(S), m)$ is consistent with the classic behavior: if $\mathbb{N}(v) = P$, then $cov(v(S), m)$ asks the coverability of $m$ in the marked Petri Net $v(S)$.

We recall that in postT-PPNs, universal coverability is true iff $cov(0(S), m)$. In a similar manner to preT-PPN, we extend the notation of $cov$, as follows: given a postT-PPN and an extended valuation $v$, we extend the predicate $cov(v(S), m)$ to extended valuations as follows: $\neg cov(v(S), m) \overset{def}{=} \forall v' \in \mathbb{N}^{\omega(v)}, \neg cov(v'(v|_{\mathbb{N}(v)}(S)), m)$. This definition is similar to the definition extended for preT-PPNs where coverability has been replaced by non-coverability.

With those extended notations, we now wonder if it is possible to compute a finite basis of $CV(S, m)$ where $S$ is a preT-PPN or a postT-PPN and $m$ a goal marking. To this end, we suggest to use an algorithm by Valk and Jantzen [20] to compute a finite representation of those sets. Nevertheless, to ensure that this algorithm is suitable to our context and that those basis are effectively computable, we need to clarify two points:

- First, this algorithm is used to compute bases of upward closed sets.
- Second, the necessary and sufficient condition recalled in Lemma 2 must be satisfied.

To address the first point, by Corollary 10 notice that in the case of postT-PPNs, the set $CV(S, m)$ is upward closed, and the procedure could be applied directly on it. In the case of
preT-PPNs, since $CV(S,m)$ is downward closed, we need to consider $\neg CV(S,m)$, which is upward closed as recalled in Section 2.3. We also recalled in that section that given a finite basis of an upward closed set, it is possible to compute a finite basis of its complement. It is therefore equivalent to being able to compute a finite basis of the set of valuations for which it is not possible to cover $m$ or to be able to compute a finite basis of the set of valuations for which it is possible to cover $m$. Thus, a finite basis of $\neg CV(S,m)$ is effectively computable iff a finite basis of $CV(S,m)$ is computable. Considering this reasoning, we address the second point through the following Lemma:

**Lemma 11.** The Valk and Jantzen condition can be reduced to the following criteria:

1. we can compute a finite representation of the coverability synthesis set in preT-PPNs iff universal coverability is decidable in preT-PPNs
2. we can compute a finite representation of the coverability synthesis set in postT-PPNs iff existential coverability is decidable in postT-PPNs

We now start by focusing on universal coverability in preT-PPNs.

## 4 Universal Coverability in preT-PPNs

We address the problem of universal coverability through that of the more general universal simultaneous unboundedness. We will prove that both are $\text{ExpSpace}$-complete.

### 4.1 Simultaneous Unboundedness

**Definition 12 (Simultaneous Unboundedness [8]).** Given $\mathcal{N} = (P,T,Pre,Post)$, and $S = (\mathcal{N},m_0)$, considering a subset $X \subseteq P$, $S$ is simultaneous $X$ unbounded if for any $B \geq 0$, there is a run $w$ such that $m_0 \xrightarrow{w} m$ and for all $i \in X$, we have $m(i) \geq B$. If $X$ is reduced to a singleton $\{p\}$, $S$ is said $p$-unbounded.

**Remark.** Notice that coverability can be easily reduced to simultaneous unboundedness by the use of an observer as depicted in Figure 2. The transition $t_{\text{obs}}$ has an input condition equal to the marking we want to cover $m$. Its output effect provides a token in a place $p_{\text{obs}}$, that, once is marked, can become unbounded through the firing of $t_{\text{cumul}}$. With this construction, $m$ is coverable in the net iff it is simultaneous $p_{\text{obs}}$-unbounded.

Since there exist polynomial translations from VASS to VAS and PN and from PN to VAS (and VASS) [18, 2], we have the following Theorem, initially stated with VASS in [8].

**Theorem 13 ([8]).** Simultaneous unboundedness for PNs is $\text{ExpSpace}$-complete.

### 4.2 Notion of Incremental Model

To prove the decidability of universal coverability in preT-PPNs, we will prove the decidability of universal simultaneous unboundedness. We will also prove that the latter belong to $\text{ExpSpace}$. Together with Remark of Section 3.1, we can then conclude that both problems are $\text{ExpSpace}$-complete.

Formally, given a parametric Petri net, and a set of places $X$, the parametric net is universally simultaneous $X$ unbounded iff for every possible valuation $v$ of its parameters, the $v$-instance of this net is simultaneous $X$ unbounded.

We first show that it is sufficient for a net to be simultaneous unbounded on a set of places in infinitely many instances (under uniform valuations) of the parametric Petri net to be universally simultaneous unbounded on this set of places. Indeed, for any valuation $v$, we can find a uniform valuation $k$ such that $v \leq k$ and apply Lemma 9.
Lemma 14. Given a marking $m$ and a marked preT-PPN $S$ and $X$ a subset of places of $S$, the two following propositions are equivalent:

1. $(S,m_0)$ is universally simultaneous $X$ unbounded
2. $\{k \in \mathbb{N} \mid (k(S),m_0) \text{ is simultaneous } X \text{ unbounded} \}$ is infinite

This Lemma is used for the proof of the upcoming Lemma 15. It is indeed an important result since it reduces the infinite set of valuations over which we should investigate to the infinite set of uniform valuation that is totally ordered i.e. two elements of this set are always comparable.

We can now address the problem of universal simultaneous unboundedness. To solve this problem, we reduce it to the existence of a classic Petri net built upon our parametric model satisfying an adequately chosen simultaneous unboundedness property. The classic Petri net is in fact obtained by evaluating a preT-PPN, called incremental net, under the uniform valuation $0$. The incremental net has a polynomial size in the original preT-PPN and it directly depends on the original preT-PPN and a sequence of distinct parametric transitions. This Section is thus driven by the idea that universal simultaneous $X$ unboundedness on a preT-PPN $S$ is equivalent to the existence of a sequence $\sigma$ of distinct parametric transitions of $S$, such that the incremental model build on $S$ and $\sigma$ evaluated under $0$ satisfies a simultaneous unboundedness property depending on $X$ and $\sigma$.

Before providing the theoretical definition, let us consider the main intuition of our construction. If a net is universally simultaneous unbounded on a set of places $X$, two main cases are possible: we can either find a path such that the places of $X$ are unbounded without using any parametric transition, and then the corresponding run works for any valuation, or we need at least one parametric transition.

In the latter case, since there is an infinite number of valuations and a finite number of parametric transitions, using the pigeonhole principle, there is at least one such transition that must be used as the first parametric transition in the run for an infinite number of valuations. The input places of its parametric arcs are therefore not bounded. Thus, the valuation of the input parametric arcs of this transition is not limiting anymore since we can generate an arbitrary large amount of tokens in the corresponding places. Therefore, we will later evaluate those parameters to $0$ in order to perform the verification on a classic Petri net.

Nevertheless, we need to ensure that the set of input places of the parametric arcs are not bounded (without using that transition). This is exactly the goal of this incremental model. Indeed, once fired, we could then consider a new net where the first parametric transition can be involved as well as non parametric transitions and investigate for the newly unbounded places. Either we can unbound the places of the goal set $X$ or we can reuse previous reasoning and choose a new parametric transition that has to be involved in infinitely many instances. What is important to note here is that at each firing of a new parametric transition, that never occurred in the run, we need to ensure that its input places of parametric input arcs were unbounded using only previous transitions of the run and to remember what are the new places that can be unbounded through the use of this new transition. We will now formalize how it is possible to remember the boundedness of the input places of parametric arcs by presenting exhaustively the model of incremental nets.

Given a preT-PPN $N = (P,T',Pre,Post,P)$ and a partition of its transitions $T' = T \cup \Theta$ between its plain and parametric transition, we denote by $N_p$ the Petri Net obtained from

\[^2\] Note that any other finite valuation would be suitable since the input places of the parametric arcs are unbounded.
between the

\( T = m, |P| = n \) and \(|\sigma| = k\). We define the incremental model \( I \) of \( N \) along \( \sigma \). We write \( I = \text{incr}(N, \sigma) \) to denote this preT-PPN. This model is illustrated by the example\(^3\) at the right hand side of Figure 3. Its construction consists of the following main steps:

(i) Consider \( N_p \) and \( k \) copies of \( N_p \), where to each of those \( k + 1 \) subnets is associated a global lock place, ensuring that exactly one copy is active at any given instant. The copies correspond to the black subnet of this example, whereas the global locks \( p_0^{\\text{\tiny incr}} \)'s are depicted in blue and dotted arcs.

(ii) Add a copy of the first \( i \) transitions of \( \sigma \) to each \( i^{\text{th}} \) copy of \( N_p \), for \( 1 \leq i \leq k \).

(iii) Between the \( i - 1^{\text{th}} \) and \( i^{\text{th}} \) subnets, add a copy of the \( i + 1^{\text{th}} \) transition of \( \sigma \), depicted in plain green arcs in Figure 3, for \( 1 \leq i \leq k \). Notice that this copy presents a special behaviour: its input effect impacts the \( i - 1^{\text{th}} \) subnet and its lock \( p_0^{\text{\tiny incr}} \) whereas its output effect impacts the \( i^{\text{th}} \) subnet and its lock \( p_0^0 \). This ensures that we change of active subnet only after the firing of a the first occurrence of a precise parametric transition.

(iv) Finally, we ensure that given every copy of a transition, including the intermediate copies that allow to change the active copy, it modifies simultaneously the places in the associated copy as explained above, but also all copies of greater index (i.e. those that have not been activated yet). Those arcs ensure that every later subnet always has the “same” marking as the active copy. They are depicted by dashed red arcs in Figure 3. Note that we synchronise the different copies and do not merge them because we use them to remember the order in which the different input places to parametric transitions become unbounded.

Let us suppose we evaluated this incremental model in order to perform an execution. At the beginning of any execution, given a precise subnet, let us say the \( i^{\text{th}} \) subnet, it follows the behavior of the subnets with lower index because of synchronisations introduced in item (iv). Then, once this copy become active, after the firing of a given parametric transition introduced in item (iii), it will dictate the behavior of the global net (and thus of the subnets with greater index through synchronisations). Once the next copy becomes active, our original \( i^{\text{th}} \) subnet cannot change its state anymore. It is now literally an historic state of the global run of the incremental net.

More formally, the incremental net \( \text{incr}(N, \sigma) \) is the preT-PPN \( \langle P, T, \text{Pre}, \text{Post}, \text{Post} \rangle \) such that \( P = \{ p_j^i \mid 0 \leq i \leq k \land 0 \leq j \leq n \} \) where \( p_j^0 \) represents the lock related to the \( i^{\text{th}} \) copy of \( N \) whereas \( p_j^i \) with \( j > 0 \) represents the copy of the place \( p_j \) of \( P \) in the \( i^{\text{th}} \) subnet and \( T = \{ t_j^i \mid 0 \leq i \leq k \land 1 \leq j \leq m \} \cup \{ \theta_j^i \mid 1 \leq j \leq i \leq k \} \cup \{ \cup_{0 \leq i \leq j} \{ \theta_j^0 \} \} \) where \( t_j^i \) represents the copy of the transition \( t_j \) of \( T \) in the \( i^{\text{th}} \) subnet, \( \theta_j^i \) represents the copy of the transition \( \theta_j \) of \( \Theta \) in the \( i^{\text{th}} \) subnet, \( \theta_j^0 \) represents a copy of the transition \( \theta_j \) from \( \sigma \) which is used to change the active copy (from the \( i - 1^{\text{th}} \) to the \( i^{\text{th}} \)).

We define with this construction a net mapping to relate places and transitions from both models. Given the two nets \( \hat{N} \) and \( I \) defined as above, considering previous notations, for each \( 0 \leq i \leq |\sigma| \) we define the application \( f_{\hat{N} \to I} \) that links the original net \( \hat{N} \) to its \( i^{\text{th}} \) copy in \( I \) (except the corresponding lock). We define \( f_{\hat{N} \to I} : T \cup \{ \theta_j \in \sigma \mid j \leq i \} \cup P \to T \cup P \) such that for \( t_j \in T \) (resp. \( \theta_j \in \sigma \) and \( p_j \in P \) ), \( f_{\hat{N} \to I}(t_j) = t_j^i \) (resp. \( f_{\hat{N} \to I}(\theta_j) = \theta_j^i \) and \( f_{\hat{N} \to I}(p_j) = p_j^i \)). We can then define \( f^{-1} \) the function that maps components of the copies

\(^3\) The exact meaning of the notations used to refer to the different components of this example will be provided after this informal intuition on the construction.
of \( \mathcal{N} \) in \( \mathcal{I} \) to their original fiber by the previous application. Formally, \( f^{-1} \) is defined by:
\[
f^{-1} : \bigcup_{0 \leq i \leq k} \text{im}(f^i_{\mathcal{N} \rightarrow \mathcal{I}}) \rightarrow \mathcal{T} \cup \mathcal{P} \cup \Theta
\]
and associates to \( t_j^i \in \mathcal{T} \) (resp. \( \theta_j^i \in \mathcal{T} \) and \( p_j^i \in \mathcal{P} \)) \( f^{-1}(t_j^i) = t_j \) (resp. \( f^{-1}(\theta_j^i) = \theta_j \) and \( f^{-1}(p_j^i) = p_j \)). Finally, we define the application \( h_{\mathcal{I} \rightarrow \mathcal{N}} : \bigcup_{0 \leq i \leq k} \{ \theta_0^i \} \subseteq \mathcal{T} \rightarrow \Theta \) that maps the intermediate parametric transitions between each copy of \( \mathcal{N} \) in \( \mathcal{I} \), \( \theta_0^i \) to their original fiber from \( \mathcal{N} \) and occurring in \( \sigma \), that is to say the \( i^{th} \) transition of \( \sigma \).

Those applications allow us to define formally the functions \( \text{Pre} \) and \( \text{Post} \). Given \( i' \) and \( j' \), let \( x_{ij} \) denote either \( t_j^i \) or \( \theta_j^i \) from \( \{ t_j^i \mid 0 \leq i \leq k \land 1 \leq j \leq m \} \cup \{ \theta_j^i \mid 1 \leq j \leq i \leq k \} \) in the following expressions:

\[
\text{Pre}(p_j^i, x_{ij}) = \begin{cases} 0 & \text{if } (i < i') \text{ or } (i > i' \text{ and } j = 0) \\ 1 & \text{if } i = i' \text{ and } j = 0 \\ \text{Pre}(f^{-1}(p_j^i), f^{-1}(x_{ij})) \text{ otherwise} \end{cases}
\]

\[
(\text{resp. } \text{Post}(p_j^i, x_{ij})) \text{ otherwise}
\]

\[
\text{Pre}(p_j^i, \theta_j^i) = \begin{cases} 0 & \text{if } (i + 1 < i') \text{ or } (i + 1 > i' \text{ and } j = 0) \\ 1 & \text{if } i + 1 = i' \text{ and } j = 0 \\ \text{Pre}(f^{-1}(p_j^i), h^{-1}(\theta_j^i)) \text{ otherwise}\end{cases}
\]

\[
\text{Post}(p_j^i, \theta_j^i) = \begin{cases} 0 & \text{if } (i < i') \text{ or } (i > i' \text{ and } j = 0) \\ 1 & \text{if } i = i' \text{ and } j = 0 \\ \text{Post}(f^{-1}(p_j^i), h^{-1}(\theta_j^i)) \text{ otherwise}\end{cases}
\]

Given a net \( \mathcal{N} \) and the function \( f^i_{\mathcal{N} \rightarrow \mathcal{I}} \) we extend the definition of \( f^i_{\mathcal{N} \rightarrow \mathcal{I}} \) to sets by \( f^i_{\mathcal{N} \rightarrow \mathcal{I}}(X) = \{ f^i(x) \mid x \in X \} \) and nets by defining \( f^i_{\mathcal{N} \rightarrow \mathcal{I}}(\mathcal{N}) \) as \( (f^i(\mathcal{P}), f^i(\mathcal{T}), \text{Pre}(f^i(\mathcal{P}) \times f^i(\mathcal{T})), \text{Post}(f^i(\mathcal{P}) \times f^i(\mathcal{T})), \mathcal{P}) \). When the context is clear, we omit the subscript \( \mathcal{N} \rightarrow \mathcal{I} \). As examples, \( f^0(\mathcal{N}) \) and \( f^i(\mathcal{N}) \) are provided in Figure 3. Finally, we associate to a marking \( m \) of \( \mathcal{N} \) the marking \( f(m) \) defined by for all \( p \in \bigcup_{0 \leq i \leq k} \text{im}(f^i(\mathcal{P})) \), \( f(m)(p) = m(f^{-1}(p)) \). Notice that this ignores the locks introduced in the net. Given the initial marking \( m_0 \), we thus define the initial marking of the incremental net \( \mu_0 \) as \( f(m_0) \) for the copies of the places, and 0 in all locks except the first one which receives 1. Formally, \( \mu_0(p_j^i) = m_0(p_j) \) if \( j \neq 0 \), 1 if \( i = j = 0 \) and 0 otherwise.

The idea behind this construction is double. First, we can enforce the order of the first occurrence of a parametric transition which is dictated by the sequence \( \sigma \). Second, we can access the exact amount of tokens stored in a place before the firing of the first occurrence of a parametric transition and thus keep an historic of the state of a run, just before the
firing of this parametric transition, through the copies of the original net. Based on those two observations, we will be able to observe if the input places of the parametric arcs of the first occurrence of a parametric transition in a run are bounded or not.

4.3 Complexity of Universal Simultaneous Unboundedness

We will now see that universal simultaneous unboundedness can be reduced to the existence of a sequence $\sigma$ of distinct parametric transitions such that the incremental net built upon this sequence is simultaneous unbounded on an adequately defined set of places. We first provide the intuition behind this statement before providing its formal version. We must ensure that each input place of a parametric arc of the first occurrence of a parametric transition is unbounded. Based on the previous construction, one can notice that given a parametric transition $\theta_i$ occurring in $\sigma$, its input places are only impacted by the transitions occurring in the first $i$ copies of the net. Thus, once $\theta_i$ is fired in the incremental net, the new feasible transitions will not impact the amount of tokens stored in the $i$ first subnets. We will thus be able to verify if the input places were bounded or not before its firing, by observing the places of the copy that occurs just before the first firing of this transition. For each parametric transition of $\sigma$, we should thus verify that the input places in the corresponding copies are unbounded, and finally verify that the places that should be unbounded as part of the original property are indeed unbounded in the last copy of the net and that for each instance of the incremental net under a uniform valuation. Nevertheless, since the corresponding input places of parametric transitions are unbounded, it is sufficient to verify this property for only one instance of the incremental net, and in particular we will later choose the $0$-instance. Indeed, if such a property is verified for any $k$-instance, then, it could be verified for any $k'$-instance (with $k' > k$) by exhibiting the witness run and performing more loops.

Lemma 15. Let $\mathcal{N} = (P, T, Pre, Post, \mathbb{P})$ be a pre$T$-PPN, such that $T' = T \cup \Theta$ where $\Theta$ represents the parametric transitions of $\mathcal{N}$ and $T$ its plain transitions. For every a set of places of $X \subseteq P$, the following propositions are equivalent:

1. $(\mathcal{N}, m_0)$ is universally simultaneous $X$ unbounded
2. $\exists \sigma = t_1, \ldots, t_l \in Pref(S_0)$, considering the incremental model $\mathcal{I}$ of $\mathcal{N}$ along $\sigma$, $\mathcal{I} = \text{incr}(\mathcal{N}, \sigma)$, $\exists k \in \mathbb{N}$, $(k(I), \mu_0)$ is simultaneous $Y$ unbounded where $Y = f_{\mathcal{N} \rightarrow \mathcal{I}}(X) \cup (\bigcup_{i \in \sigma} f_{\mathcal{N} \rightarrow \mathcal{I}}^{-1}(\Pi(t_i)))$.
3. $\exists \sigma = t_1, \ldots, t_l \in Pref(S_0)$, considering the incremental model $\mathcal{I}$ of $\mathcal{N}$ along $\sigma$, $\mathcal{I} = \text{incr}(\mathcal{N}, \sigma)$, $\forall k \in \mathbb{N}$, $(k(I), \mu_0)$ is simultaneous $Y$ unbounded where $Y = f_{\mathcal{N} \rightarrow \mathcal{I}}^l(X) \cup (\bigcup_{i \in \sigma} f_{\mathcal{N} \rightarrow \mathcal{I}}^{-1}(\Pi(t_i)))$.

Following the notations, $Pref(S_\Theta)$ corresponds to the finite set of sequences of distinct parametric transitions. Remark that $f_{\mathcal{N} \rightarrow \mathcal{I}}(X)$ represents the copy of the places of $X$ in the last subnet of $\mathcal{I}$. The set $\bigcup_{t_i \in \sigma} f_{\mathcal{N} \rightarrow \mathcal{I}}^{-1}(\Pi(t_i))$ is a bit more complex: for each transition $t_i \in \sigma$, $\Pi(t_i)$ represents the input places of the parametric arcs. We therefore address here the unboundedness of the copies of these places in the corresponding subnet of the $\mathcal{I}$.

Proof. We provide here the sketch of the implication $(1) \Rightarrow (3)$. The goal is to find the sequence of parametric transitions along which we construct the incremental model seen in Section 4.2. This proof is done by induction on the number of parametric transitions in $\mathcal{N}$. In the base case, $\mathcal{N}$ is a PN. Therefore the incremental model considered is isomorphic to $\mathcal{N}$ and the property is immediate.
In the inductive step, the case where it is possible that the places from $X$ are unbounded without using parametric transitions is straightforward. In the other case, we show that there is a parametric transition $\theta$ that can be used as the first parametric transition occurring in a run leading to some simultaneous $X$ unbounded markings in the coverability tree of $(k(N), m_0)$ for an infinite number of uniform valuation $k$.

1. We then prove that the input places of the parametric input arcs of $\theta$ must be unbounded in $(N_p, m_0)$, that is to say, there is a marking $z$ with some $\omega$’s on those components in the basis of the coverability set of this net.

2. We now consider the net where those places have been removed and the projection of the marking obtained by firing $\theta$ from $z$. In this net $\theta$ is then a plain transition. Using the induction assumption, we can build an incremental model $J$ along a sequence $\sigma'$ for this reduced net.

3. The end of this proof consists in building the incremental model $I$ of $N$ along the sequence $\theta\sigma'$ and to construct the set $Y$. There is no particular difficulty in this last point but the construction is a bit involved.

From Lemma 15 we can observe that answering the universal simultaneous $X$ unboundedness on a preT-PPN can be reduced to guessing an element $\sigma$ of the finite set $Pref(S_\Theta)$ such that $(0(I), \mu_0)$ is simultaneous $Y$ unbounded. Since checking simultaneous $X$ unboundedness can be done in EXPSPACE as recalled in Theorem 13, we obtain a NEXPSPACE procedure. Then, a well-known theorem by Savitch [19] stating that there is therefore a EXPSPACE deterministic procedure answering this problem and the Remark from Section 3.1 allow us to deduce Theorem 16. Note that the following Corollary directly comes from Theorem 16 and Lemma 11.

\begin{theorem}
The Universal Simultaneous Unboundedness problem for preT-PPNs is EXPSPACE-complete.
\end{theorem}

\begin{corollary}
Given a marked preT-PPN $S$ and a marking $m$, we can compute a finite representation of $CV(S, m)$.
\end{corollary}

5 Synthesis in postT-PPNs and distinctT-PPNs

5.1 Complexity of Existential Coverability in postT-PPNs

We propose here a cleaner proof for the decidability of the existential coverability in postT-PPNs, and provide its complexity. We use a polynomial time transformation\footnote{More specifically we obtain an $\omega$-output-PN or $\omega$OPN for short, which corresponds to the natural subclass of $\omega$PNs where $Pre \in P \times T \rightarrow \mathbb{N}$.} from postT-PPN to $\omega$PN (see [13]) which preserves existential coverability and invoke a transformation from $\omega$PN to PN underlined in [13]. First, we recall definitions and results from [13].

\begin{definition}[$\omega$-Petri Net [13]]
An $\omega$-Petri Net ($\omega$PN) is a tuple $(P, T, Pre, Post)$ where $P$ and $T$ are respectively a finite set of places and transitions. $Pre$ (resp. $Post$) is a function of $P \times T$ to $\mathbb{N}_\omega$ that gives the input (resp. output) effect of a transition $t$ on a place $p$.
\end{definition}

\begin{definition}[$\omega$PN Semantics]
Given a marking $m$, and a transition $t$ such that $m \geq Pre(t)$, firing $t$ from $m$ gives a new marking $m'$ s.t. $\forall p \in P, m'(p) = m(p) - Pre(p, t) + o$
\end{definition}
Coverability Synthesis in PPNs

where \( o = \text{Post}(t, p) \) if \( \text{Post}(p, t) \in \mathbb{N} \) and \( o \geq 0 \) if \( \text{Post}(p, t) = \omega \). We denote this by \( m \xrightarrow{l} m' \). Thus \( \text{Post}(p, t) = \omega \) means that an arbitrary number of tokens are generated in \( p \).

From this semantics, we can notice that \( \omega \)'s play a role not unlike output parameters, with the crucial difference that their “value” can change along the execution of the net. Let consider a postT-PPN \( N \) and let us associate to this model the \( \omega \)OPN \( N' \) such that we replace each parametric arc of \( N \) by an \( \omega \) arc.

Lemma 20 (postT-PPNs to \( \omega \)OPNs). Let \( N \) be a postT-PPN (which involves parameters of a set \( P \)) and let \( N' \) be its corresponding \( \omega \)OPN (with the same set of places and transitions) and let \( m_0 \) be their common initial marking. Given a marking \( m \in \mathcal{RS}(N', m_0) \), there exists a valuation \( v \) such that there exists a marking \( m' \geq m \) with \( m' \in \mathcal{RS}(v(N), m_0) \). Moreover\(^5\), \( \bigcup_{v \in \mathcal{FR}} \mathcal{RS}(v(N'), m_0) \subseteq \mathcal{RS}(N', m_0) \).

We can thus directly deduce Theorem 21 by reducing existential coverability in postT-PPNs to coverability in \( \omega \)PN which belongs to \( \text{ExpSPACE} \) by [13]. Note that the following Corollary comes from Theorem 21 and Lemma 11.

Theorem 21 (Complexity of Existential Coverability). The existential coverability problem on postT-PPNs is \( \text{ExpSPACE} \)-complete.

Corollary 22. Given a marked postT-PPN \( S \) and a marking \( m \), we can compute a finite representation of \( \text{CV}(S, m) \).

5.2 Representing the Coverability Synthesis Set for DistinctT-PPNs

Let us finally consider the case of PPNs in which the set of parameters used as input weights, and the set of parameters used as output weights, are disjoint. For this class, called distinctT-PPNs, the emptiness of the solution set to the synthesis problem for coverability is decidable [7]. Interestingly, we can adapt an idea originally used for L/U-automata in [15] to prove that the structure of this set is however much more complex than for preT-PPNs or postT-PPNs. In particular, one cannot represent this set with a finite set, a finite union of downward and/or upward closed sets or a finite union of convex polyhedra.

Lemma 23. If it can be computed, the solution of the synthesis of coverability in distinctT-PPN cannot, in general, be represented using any formalism for which emptiness of the intersection with equality constraints is decidable.

6 Conclusion

It can be challenging to find meaningful parametric infinite state systems with decidable decision problems. We achieved here to prove a powerful result for two strict syntactical subclasses of parametric Petri nets: interestingly, the set of all valid valuations of parameters, allowing to cover a given marking, is effectively computable for parametric Petri nets where parameters are restricted to only input arcs or only output arcs.

Indeed, we have shown how the computability of the synthesis set for coverability in preT-PPNs and postT-PPNs can be reduced to a decision problem, respectively, universal coverability and existential coverability, which is then used in Valk and Jantzen’s procedure.

\(^5\) Notice that this is only an inclusion. Indeed, contrarily to postT-PPNs, in \( \omega \)PNs, the effect of an arc can change along the same execution.
We proved that these two decision problems are both EXPSPACE-complete. Putting the two types of parameters together while forbidding any parameter to be used as both an input and output weight preserves the decidability of the emptiness of the solution set. However, we have proved that, even with this restriction, the solution set can in general not be represented using any formalism for whichemptiness of the intersection with equality constraints is decidable, which seems a big restriction in practice.

Future work includes studying (simultaneous) unboundedness for classes other than preT-PPNs that is to say postT-PPNs and distinctT-PPNs. Most problems (such as universal simultaneous unboundedness for postT-PPNs and distinctT-PPNs) can be settled easily by adapting the proofs of [7], except for existential simultaneous unboundedness for postT-PPNs. The translation to \(\omega\)PNs proposed here is not sufficient to conclude its decidability either since we have to ensure that the increasing markings are all reached for a common parameter valuation.

Acknowledgements. We are grateful to anonymous commenters whose helpful remarks have allowed us to improve this paper.

References

Coverability Synthesis in PPNs

Divide and Congruence III: Stability & Divergence

Wan Fokkink¹, Rob van Glabbeek², and Bas Luttik³

¹ Vrije Universiteit Amsterdam, The Netherlands
² Data61, CSIRO, Sydney, Australia and University of New South Wales, Sydney, Australia
³ Eindhoven University of Technology, The Netherlands

Abstract

In two earlier papers we derived congruence formats for weak semantics on the basis of a decomposition method for modal formulas. The idea is that a congruence format for a semantics must ensure that the formulas in the modal characterisation of this semantics are always decomposed into formulas that are again in this modal characterisation. Here this work is extended with important stability and divergence requirements. Stability refers to the absence of a \( \tau \)-transition. We show, using the decomposition method, how congruence formats can be relaxed for weak semantics that are stability-respecting. Divergence, which refers to the presence of an infinite sequence of \( \tau \)-transitions, escapes the inductive decomposition method. We circumvent this problem by proving that a congruence format for a stability-respecting weak semantics is also a congruence format for its divergence-preserving counterpart.

1998 ACM Subject Classification F.3.2 Operational Semantics, F.4.1 Modal Logic

Keywords and phrases Structural Operational Semantics, Weak Semantics, Modal Logic

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2017.15

1 Introduction

Structural operational semantics generates a labelled transition system, in which states are the closed terms over a signature, and transitions between states carry labels. Transitions are obtained from a transition system specification (TSS), consisting of proof rules called transition rules. States in labelled transition systems can be identified by a wide range of behavioural equivalences, based on e.g. branching structure or decorated versions of execution sequences. Weak semantics, which take into account the internal action \( \tau \), are classified in [11]. A significant number of the weak semantics based on a bisimulation relation carry a stability or divergence requirement. Stability refers to the absence of a \( \tau \)-transition and divergence to the presence of an infinite sequence of \( \tau \)-transitions.

In general a behavioural equivalence induced by a TSS is not guaranteed to be a congruence, i.e. the equivalence class of a term \( f(p_1,\ldots,p_n) \) need not be determined by \( f \) and the equivalence classes of its arguments \( p_1,\ldots,p_n \). Being a congruence is an important property, for instance in order to fit the equivalence into an axiomatic framework. Respecting stability or preserving divergence sometimes needs to be imposed in order to obtain a congruence relation, for example in case of the priority operator [1].

Modal logic captures observations an experimenter can make during a session with a process. A modal characterisation of an equivalence on processes consists of a class \( C \) of modal formulas such that two processes are equivalent if and only if they satisfy the same formulas in \( C \). For instance, Hennessy-Milner logic [17] constitutes a modal characterisation of (strong) bisimilarity. A cornerstone for the current paper is the work in [3] to decompose formulas from Hennessy-Milner logic with respect to a structural operational semantics in
the ntyft format [16] without lookahead. Here the decomposition of a modal formula $\varphi$ w.r.t. a term $t = f(p_1, \ldots, p_n)$ is a selection of $n$-tuples of modal formulas, one of which needs to be satisfied by the processes $p_i$ in order for $t$ to satisfy $\varphi$. Based on this method, formats for behavioural equivalences can be generated from their modal characterisation, ensuring that they are congruences. Such formats help to avoid repetitive congruence proofs and obtain insight into the congruence property. Key idea is that congruence is ensured if formulas from the modal characterisation $C$ of an equivalence are always decomposed into formulas that are again in $C$. This approach was extended to weak semantics in [10, 7].

Here we expand the latter work to weak semantics that respect stability or preserve divergence. We focus on branching bisimilarity and rooted branching bisimilarity [15] and consider for each a stability-respecting and two divergence-preserving variants. Divergence-preserving branching bisimilarity [15] is the coarsest congruence relation for the parallel composition operator that only equates processes satisfying the same formulas from the well-known temporal logic CTL* minus the next-time operator $X$ [14]. With regard to stability the expansion is relatively straightforward: we extend the modal characterisation of the semantics with one clause to capture that a semantics is stability-respecting, and study the decomposition of this additional clause. Next we show how the congruence formats for branching bisimilarity and rooted branching bisimilarity from [10] can be relaxed, owing to the extended modal characterisation for stability-respecting branching bisimilarity. Notably, the transition rules for the priority operator are within the more relaxed formats. The divergence preservation property escapes the inductive decomposition method, as it concerns an infinite sequence of $\tau$-transitions. We overcome this problem by a general framework for lifting congruence formats from a weak semantics $\approx$ to a finer semantics $\sim$. We show four applications of this method. In two cases $\approx$ is stability-respecting and in two cases rooted stability-respecting branching bisimilarity, while in $\sim$ stability is replaced by two different forms of divergence. Hence the congruence format for stability-respecting branching bisimilarity is also applicable to divergence-preserving as well as weakly divergence-preserving branching bisimilarity; and likewise for their rooted counterparts.

2 Preliminaries

2.1 Stability-respecting / divergence-preserving branching bisimilarity

A labelled transition system (LTS) is a triple $(P, Act, \rightarrow)$, with $P$ a set of processes, $Act$ a set of actions, and $\rightarrow \subseteq P \times Act \times P$. We normally let $Act = A \cup \{\tau\}$ where $\tau$ is an internal action and $A$ some set of external or observable actions not containing $\tau$. We write $A_\tau$ for $A \cup \{\tau\}$. We use $p, q$ to denote processes, $\alpha, \beta, \gamma$ for elements of $A_\tau$, and $a, b$ for elements of $A$. We write $p \xrightarrow{\alpha} q$ for $(p, \alpha, q) \in \rightarrow$, $p \xrightarrow{\alpha}$ for $\exists q \in P : p \xrightarrow{\alpha} q$, and $p \xrightarrow{\alpha}$ for $\neg(p \xrightarrow{\alpha})$. Furthermore, $\tau \rightarrow$ denotes the transitive-reflexive closure of $\tau \rightarrow$.

$\blacktriangleright$ Definition 1. Let $B \subseteq P \times P$ be a symmetric relation.

- $B$ is a branching bisimulation if $p B q$ and $p \xrightarrow{\alpha} p'$ implies either $\alpha = \tau$ and $p' B q$, or $q \xrightarrow{\alpha} q''$ for some $q''$ with $p B q'$ and $p' B q''$.
- $B$ is stability-respecting if $p B q$ and $p \xrightarrow{\tau}$ implies $q \xrightarrow{\tau} q'$ for some $q'$ with $p B q'$.
- $B$ is divergence-preserving if it satisfies the following condition:

  (D) if $p B q$ and there is an infinite sequence of processes $(p_k)_{k \in \mathbb{N}}$ with $p = p_0$, $p_k \xrightarrow{\tau} p_{k+1}$ and $p_k B q$ for all $k \in \mathbb{N}$, then there exists an infinite sequence of processes $(q_\ell)_{\ell \in \mathbb{N}}$ with $q = q_0$, $q_\ell \xrightarrow{\tau} q_{\ell+1}$ for all $\ell \in \mathbb{N}$, and $p_k B q_\ell$ for all $k, \ell \in \mathbb{N}$.

The definition of a weakly divergence-preserving relation is obtained by omitting the condition “and $p_k B q_\ell$ for all $k, \ell \in \mathbb{N}$.”
Processes $p, q$ are branching bisimilar, denoted $p \equiv_b^\tau q$, if there exists a branching bisimulation $B$ with $p \sim B q$. They are stability-respecting, divergence-preserving or weakly divergence-preserving branching bisimilar, denoted $p \equiv_b^\tau q$, $p \equiv_b^\Delta q$ or $p \equiv_b^{\Delta^\bot} q$, if moreover $B$ is stability-respecting, divergence-preserving or weakly divergence-preserving, respectively.

We have $\equiv_b^\tau \supset \equiv_b^\Delta \supset \equiv_b^\Delta^\bot \supset \equiv_b^\tau$. The relations $\equiv_b^\tau$, $\equiv_b^\Delta$ and $\equiv_b^{\Delta^\bot}$ are equivalences [2, 11, 13]. However, they are not congruences with respect to most process algebras from the literature, meaning that the equivalence class of a process $f(p_1, \ldots, p_n)$, with $f$ an $n$-ary function symbol, is not always determined by the equivalence classes of its arguments, i.e. the processes $p_1, \ldots, p_n$. Therefore an additional rootedness condition is imposed.

**Definition 2.** Rooted branching bisimilarity, $\equiv_b^{rb}$, is the largest symmetric relation on $P$ with $p \equiv_b^{rb} q$ and $p \alpha \rightarrow p' \text{ implies } q \alpha \rightarrow q'$ for some $q'$ with $p' \equiv_b^\tau q'$. Likewise, rooted stability-respecting, divergence-preserving or weakly divergence-preserving branching bisimilarity, denoted by $p \equiv_b^{rb} q$, $p \equiv_b^{rb} q$ or $p \equiv_b^{rb} q$, is the largest symmetric relation $R$ on $P$ with $p R q$ and $p \alpha \rightarrow p' \text{ implies } q \alpha \rightarrow q'$ for some $q'$ with $p' \equiv_b^{rb} q'$, $p' \equiv_b^{rb} q'$ or $p' \equiv_b^{rb} q'$.

The various notions of bisimilarity defined above are examples of so-called behavioural equivalences. For a general formulation of the results in Section 4, it is convenient to formally define a notion of behavioural equivalence that includes at least the examples above. Note that a common feature of their definitions is that they associate with every LTS $G = (P_G, Act_G, \rightarrow_G)$ a binary relation $\sim_G$. (For instance, in the case of branching bisimilarity, the relation $\sim_G$ associated with $G$ is defined as the binary relation $\equiv_b \subseteq P_G \times P_G$ such that $p \equiv_b^\tau q$ if (and only if) there exists a branching bisimulation $B \subseteq P_G \times P_G$ with $p B q$.) One may, thus, think of a behavioural equivalence as a family of binary relations indexed by LTSs. It turns out that we need to impose just one extra condition on such families to arrive at a suitable formalisation of the notion of behavioural equivalence. The condition states that the relation associated with the disjoint union of two LTSs restricted to one of the components coincides with the relation associated with that component.

Two LTSs $G = (P_G, Act_G, \rightarrow_G)$ and $H = (P_H, Act_H, \rightarrow_H)$ are called disjoint if $P_G \cap P_H = \emptyset$. In that case $G \upharpoonright H$ denotes their union $(P_G \cup P_H, Act_G \cup Act_H, \rightarrow_G \cup \rightarrow_H)$.

A behavioural equivalence $\sim$ on LTSs is a family of equivalence relations $\sim_G$, one for every LTS $G$, such that for each pair of disjoint LTSs $G = (P_G, Act_G, \rightarrow_G)$ and $H = (P_H, Act_H, \rightarrow_H)$ we have $g \sim g' \Leftrightarrow g \sim_H g'$ for any $g, g' \in P_G$. The notions of bisimilarity defined above clearly qualify as behavioural equivalences. We write $\sim \subseteq \approx$ iff $\sim_G \subseteq \approx_G$ for each LTS $G$.

### 2.2 Modal logic

Modal logic formulas express behavioural properties of processes. Following [11], we extend Hennessy-Milner logic [17] with connectives $\langle i \rangle \varphi$ and $\langle \tau \rangle \varphi$, expressing that a process can perform zero or more, respectively zero or one, $\tau$-transitions to a process where $\varphi$ holds.

**Definition 3.** The class $\mathcal{O}$ of modal formulas is defined as follows, where $I$ ranges over all index sets and $\alpha$ over $A_\tau$: $\varphi ::= \bigwedge_{i \in I} \varphi_i \mid \neg \varphi \mid \langle \alpha \rangle \varphi \mid \langle i \rangle \varphi \mid \langle \tau \rangle \varphi$. We write $\top$ for the empty conjunction, $\varphi_1 \land \varphi_2$ for $\bigwedge_{i \in \{1,2\}} \varphi_i$, $\varphi_1 \langle \alpha \rangle \varphi_2$ for $\varphi_1 \land \langle \alpha \rangle \varphi_2$, and $\varphi \langle \tau \rangle \varphi'$ for $\varphi \land \langle \tau \rangle \varphi'$. $p \models \varphi$ denotes that process $p$ satisfies formula $\varphi$. The first two operators represent the standard Boolean operators conjunction and negation. By definition, $p \models \langle \alpha \rangle \varphi$ if $p, s \models \varphi$ for some $p'$ with $p' \models \varphi$, $p \models \langle i \rangle \varphi$ if $p, s \models \varphi$ for some $p'$ with $p' \models \varphi$, and $p \models \langle \tau \rangle \varphi$ if either $p \models \varphi$ or $p, s \models \varphi$ for some $p'$ with $p' \models \varphi$. 

*CONCUR 2017*
For each $L \subseteq \mathcal{O}$, we write $p \sim_L q$ if $p$ and $q$ satisfy the same formulas in $L$. We say that $L$ is a modal characterisation of some behavioural equivalence $\sim$ if $\sim_L$ coincides with $\sim$. We write $\varphi \equiv \varphi'$ if $p \models \varphi \iff p \models \varphi'$ for all processes $p$. The class $L^\equiv$ denotes the closure of $L \subseteq \mathcal{O}$ under $\equiv$. Trivially, $p \sim_L q \iff p \sim_L = q$.

**Definition 4 ([11]).** The subclasses $O_b$ and $O_{rb}$ of $\mathcal{O}$ are defined as follows, where $a$ ranges over $A$ and $\alpha$ over $\mathcal{A}$:

$O_b$

$\varphi := \bigwedge_{i \in I} \forall i \mid \neg \varphi | \langle e \rangle (\varphi(\bar{a}) \varphi) | \langle e \rangle (\varphi(\bar{a}) \varphi) | \langle e \rangle (\varphi(\bar{a}) \varphi) | \langle e \rangle (\varphi(\bar{a}) \varphi) | \langle e \rangle (\varphi(\bar{a}) \varphi) | \langle e \rangle (\varphi(\bar{a}) \varphi)$

$O_{rb}$

$\varphi := \bigwedge_{i \in I} \forall i \mid \neg \varphi | \langle a \rangle \varphi | \varphi | (\varphi \in O_b)$

$O_b$ and $O_{rb}$ are modal characterisations of $\equiv_b$ and $\equiv_{rb}$, respectively (see [10]).

The idea behind stability is: (I) if $p \equiv_b^s q$ and $p \Rightarrow p' \Rightarrow q'$ then $q \not\Rightarrow q' \Rightarrow q'$. In the definition of $\equiv_b^s$ this was formulated more weakly: (II) if $p \equiv_b^s q$ and $p \not\Rightarrow q$, then $q \not\Rightarrow q \Rightarrow q'$. It is easy to see that formulations (I) and (II) are equivalent. The additional clause in the following modal characterisation of stability-respecting branching bisimilarity is based on (I).

**Definition 5 ([11]).** The subclasses $O_b^s$ and $O_{rb}^s$ of $\mathcal{O}$ are defined as follows:

$O_b^s$

$\varphi := \bigwedge_{i \in I} \forall i \mid \neg \varphi | \langle e \rangle (\varphi(\bar{a}) \varphi) | \langle e \rangle (\varphi(\bar{a}) \varphi) | \langle e \rangle (\varphi(\bar{a}) \varphi)$

$O_{rb}^s$

$\varphi := \bigwedge_{i \in I} \forall i \mid \neg \varphi | \langle a \rangle \varphi | \varphi | (\varphi \in O_b^s)$

The additional clause $\langle e \rangle (\neg \varphi \bar{a} \varphi)$ in $O_b^s$ expresses stability. The first part $\langle e \rangle (\neg \varphi \bar{a} \varphi)$ captures $p \Rightarrow p' \Rightarrow q'$, while the second part . . . $\varphi$ captures the stability-respecting branching bisimulation class of $p'$. Since $p \not\Rightarrow q$, unrooted and rooted stability-respecting branching bisimilarity coincide for $p'$, which allows us to take the second part from $O_{rb}^s$.

**Theorem 6.** $p \equiv_b^s q \iff p \sim \equiv_b^s q$ and $p \equiv_{rb}^s q \iff p \sim \equiv_{rb}^s q$, for all $p, q \in \mathcal{P}$.

All proofs omitted from the paper can be found in [8].

### 2.3 Structural operational semantics

A **signature** is a set $\Sigma$ of function symbols $f$ with arity $ar(f)$. A function symbol of arity 0 is called a **constant**. Let $V$ be an infinite set of variables; we assume $|\Sigma|, |A| \leq |V|$. A syntactic object is **closed** if it does not contain any variables. The sets $T(\Sigma)$ and $T(\Sigma)$ of terms over $\Sigma$ and $V$ and closed terms over $\Sigma$, respectively, are defined as usual; $t, u, v, w$ denote terms, $p, q$ denote closed terms, and $var(t)$ is the set of variables that occur in term $t$. A substitution $\sigma$ is a partial function from $V$ to $T(\Sigma)$. A closed substitution is a total function from $V$ to closed terms. The domain of substitutions is extended to $T(\Sigma)$ as usual.

Structural operational semantics generates an LTS in which the processes are the closed terms. The labelled transitions between processes are obtained from a transition system specification, which consists of a set of proof rules called transition rules.

A **positive** or **negative** literal is an expression $\tau \rightarrow u$ or $\tau \leftarrow v$. A **transition** rule is of the form $\frac{H}{\tau}$ with $H$ a set of literals called the **premises**, and $\lambda$ a literal called the **conclusion**; the terms at the left- and right-hand side of $\lambda$ are called the **source** and **target**. A rule $\frac{H}{\lambda}$ is also written $\lambda$. A rule is **standard** if it has a positive conclusion. A **transition system specification** (TSS), written $(\Sigma, Act, R)$, consists of a signature $\Sigma$, a set of actions $Act$, and a set $R$ of transition rules over $\Sigma$. A TSS is **standard** if all its rules are.

A TSS specifies an LTS in which the transitions are the closed positive literals that can be proved using the rules of the TSS. Since rules may have negative premises, consistency is a concern. Literals $\tau \rightarrow u$ and $\tau \leftarrow v$ are said to **deny** each other; a notion of provability...
associated with TSSs is consistent if it is not possible to prove two literals that deny each other. To arrive at a consistent notion of provability, we proceed in two steps: first we define the notion of an irredundant proof, which on a standard TSS does not allow the derivation of negative literals at all, and then arrive at a notion of well-supported proof that allows the derivation of negative literals whose denials are manifestly underivable by irredundant proofs. In [12] it was shown that the notion of well-supported provability is consistent.

Definition 7 ([3]). Let $P = (\Sigma, Act, R)$ be a TSS. An irredundant proof from $P$ of a rule $\frac{H}{X}$ is a well-founded tree with the nodes labelled by literals and some of the leaves marked “hypothesis”, such that the root has label $\lambda$, $H$ is the set of labels of the hypotheses, and if $\mu$ is the label of a node that is not a hypothesis and $K$ is the set of labels of the children of this node then $\frac{\mu}{P}$ is a substitution instance of a rule in $R$. The rule $\frac{H}{X}$ is irredundantly provable from $P$, notation $P \vdash_{irr} \frac{H}{X}$, if such a proof exists.

Definition 8 ([12]). Let $P = (\Sigma, Act, R)$ be a standard TSS. A well-supported proof from $P$ of a closed literal $\lambda$ is a well-founded tree with the nodes labelled by closed literals, such that the root is labelled by $\lambda$, and if $\mu$ is the label of a node and $K$ is the set of labels of the children of this node, then:

1. either $\mu$ is positive and $\frac{\mu}{P}$ is a closed substitution instance of a rule in $R$;
2. or $\mu$ is negative and for each set $N$ of closed negative literals with $\frac{\mu}{P}$ irredudantly provable from $P$ and $\nu$ a closed positive literal denying $\mu$, a literal in $K$ denotes one in $N$. $P \vdash_{ws} \lambda$ denotes that a well-supported proof from $P$ of $\lambda$ exists. A standard TSS $P$ is complete if for each $p$ and $\alpha$, either $P \vdash_{ws} p \overset{\alpha}{\rightarrow}$ or $P \vdash_{ws} p \overset{\alpha}{\rightarrow} q$ for some closed term $q$.

If $P = (\Sigma, Act, R)$ is a complete TSS, then the LTS associated with $P$ is $(\Sigma(\Sigma), Act, \rightarrow)$ with $\rightarrow = \{ (p, \alpha, q) \mid P \vdash_{ws} p \overset{\alpha}{\rightarrow} q \}$. We do not associate an LTS with an incomplete TSS.

2.4 Congruence formats

Let $P = (\Sigma, Act, R)$ be a transition system specification, and let $\sim_P$ be an equivalence relation defined on the set of closed terms $T(\Sigma)$. Then $\sim_P$ is a congruence for $P$ if, for each $f \in \Sigma$, we have that $p_i \sim_P q_i$ implies $f(p_1, \ldots, p_{ar(f)}) \sim_P f(q_1, \ldots, q_{ar(f)})$. Note that this is the case if for each open term $t \in T(\Sigma)$ and each pair of closed substitutions $\rho, \rho' : V \rightarrow T(\Sigma)$ we have $(\forall x \in var(t), \rho(x) \sim_P \rho'(x)) \Rightarrow \rho(t) \sim_P \rho'(t)$.

Every complete TSS generates an LTS of which the states are the closed terms of the TSS. Thus each behavioural equivalence $\sim$ associates with every TSS $P$ an equivalence $\sim_P$ on its set of closed terms. By a congruence format for $\sim$ we mean a class of TSSs such that for every TSS $P$ in the class the equivalence $\sim_P$ is a congruence. Usually, a congruence format is defined by means of a list of syntactic restrictions on the rules of TSSs.

In an ntyft rule, right-hand sides of positive premises are distinct variables that do not occur in the source. An ntyft rule is an ntyxt rule if its source is a variable, an ntyxt rule if its source contains exactly one function symbol and no multiple occurrences of variables, and an ntyxt rule if the left-hand sides of its premises are variables. A variable in a rule is free if it occurs neither in the source nor in right-hand sides of premises. A rule has lookahead if some variable occurs in the right-hand side of a premise and in the left-hand side of a premise. A rule is decent if it has no lookahead and does not contain free variables. Each combination of syntactic restrictions on rules induces a format for TSSs of the same name. For instance, a TSS is in decent ntyft format if it contains decent ntyft rules only. A TSS is in ready simulation format if it consists of ntyft and ntyxt rules that have no lookahead.
In congruence formats for weak semantics, lookahead must be forbidden. To see this, consider CCS [18] extended with an operator $f$ defined by $\vdash f(x) \rightarrow x$. Then $a \mathcal{B} b \iff a \times b \mathcal{B} a \times b$, whereas $f(ab) \not\equiv_{\mathcal{E}} f(ab) f(ab)$. Therefore congruence formats for weak semantics are generally obtained by imposing additional restrictions on the ready simulation format.

### 2.5 Decomposition of modal formulas

The decomposition method from [10] gives a special treatment to arguments of function symbols that are deemed *patient*; a predicate marks these arguments. Let $\Gamma$ be a predicate on arguments of function symbols. A standard ntyyf rule is a $\Gamma$-patient rule [6] if it is of the form $\frac{x_i \rightarrow y}{f(x_1, \ldots, x_{ar(f)}) \rightarrow f(x_1, \ldots, x_{i-1}, y, x_{i+1}, \ldots, x_{ar(f)})}$ with $\Gamma(f, i)$. A TSS is $\Gamma$-patient if it contains all $\Gamma$-patience rules. A standard ntyyf rule is $\Gamma$-patient if it is irredundantly provable from the $\Gamma$-patience rules; else it is called $\Gamma$-impatient. Let $\Gamma$ be a predicate on $\{(f, i) \mid 1 \leq i \leq ar(f), f \in \Sigma\}$. If $\Gamma(f, i)$, then argument $i$ of $f$ is $\Gamma$-liquid; otherwise it is $\Gamma$-frozen [3]. An occurrence of $x$ in $t$ is $\Gamma$-liquid if either $t = x$, or $t = f(t_1, \ldots, t_{ar(f)})$ and the occurrence is $\Gamma$-liquid in $t_i$ for a liquid argument $i$ of $f$; otherwise the occurrence is $\Gamma$-frozen.

To each term $t$ and formula $\phi$, we assign a set $t^{-1}(\phi)$ of decomposition mappings $\psi : V \rightarrow \mathbb{Q}$, such that $\rho(t) \models \phi$ iff there is a $\psi \in t^{-1}(\phi)$ with $\psi(x) \models \psi(x)$ for all $x \in \text{var}(t)$. To define $t^{-1}(\phi)$, we use a result from [3], where for each standard TSS $P$ in ready simulation format a collection of decent ntyyf rules, called $P$-ruoids, is constructed. First, $P$ is converted into a non-standard TSS $P^+$ with the property that, for all closed literals $\mu$, we have $P^+ \models \mu$ if and only if $\mu$ is irredundantly provable from $P^+$. The $P$-ruoids are the decent ntyyf rules irredundantly provable from $P^+$. In [3] it was proved that there is a well-supported proof from $P$ of a transition $\rho(t) \xrightarrow{a} q$, with $\rho$ a closed substitution, if and only if there is a proof of this transition that uses at the root a $P$-ruoid with source $t$.

**Definition 9** ([10]). Let $P = (\Sigma, A_t, R)$ be a $\Gamma$-patient standard TSS in ready simulation format. We define $\cdot^{-1} : \tau(\Sigma) \times \mathbb{Q} \rightarrow (V \rightarrow \mathbb{Q})$ as the function that for each $t \in \tau(\Sigma)$ and $\phi \in \mathbb{Q}$ returns the smallest set $t^{-1}(\phi) \subseteq \tau(V \rightarrow \mathbb{Q})$ of decomposition mappings $\psi : V \rightarrow \mathbb{Q}$ satisfying the following six conditions. Let $t$ denote a univariate term, i.e. without multiple occurrences of the same variable. (Cases 1–5 associate with every univariate term $t$ a set $t^{-1}(\phi)$.)

1. $\psi \in t^{-1}(\bigwedge_{i \in I} \phi_i)$ iff there are $\psi_i \in t^{-1}(\phi_i)$ for $i \in I$ with $\psi(x) = \bigwedge_{i \in I} \psi_i(x)$ for $x \in V$;
2. $\psi \in t^{-1}(\neg \phi)$ iff there is a function $h : t^{-1}(\phi) \rightarrow \text{var}(t)$ such that $\psi(x) = \bigwedge_{h^{-1}(x) \Gamma \chi(x)}$ if $x \in \text{var}(t)$, and $\psi(x) = \top$ otherwise;
3. $\psi \in t^{-1}(\langle \alpha \rangle \phi)$ iff there is a $\alpha$-ruoid $\frac{H}{t \rightarrow u}$ and a $\chi \in u^{-1}(\phi)$ such that $\psi(x) = \chi(x) \wedge \bigwedge_{x \rightarrow y \in H} \langle \beta \rangle \lambda(y) \wedge \bigwedge_{x \rightarrow y \in H} \neg(\gamma) \top$ if $x \in \text{var}(t)$, and $\psi(x) = \top$ otherwise;
4. $\psi \in t^{-1}(\langle e \rangle \phi)$ if one of the following holds:
   a. either there is a $\chi \in t^{-1}(\phi)$ such that $\psi(x) = \langle e \rangle \chi(x)$ if $x$ occurs $\Gamma$-liquid in $t$, and $\psi(x) = \chi(x)$ otherwise;
   b. or there is a $\Gamma$-impatient $\alpha$-ruoid $\frac{H}{t \rightarrow u}$ and a $\chi \in u^{-1}(\langle e \rangle \phi)$ such that $\psi(x) = \top$ if $x \not\in \text{var}(t)$, $\psi(x) = \langle e \rangle \left(\chi(x) \wedge \bigwedge_{x \rightarrow y \in H} \langle \beta \rangle \lambda(y) \wedge \bigwedge_{x \rightarrow y \in H} \neg(\gamma) \top\right)$ if $x$ occurs $\Gamma$-liquid in $t$, and $\psi(x) = \chi(x) \wedge \bigwedge_{x \rightarrow y \in H} \langle \beta \rangle \lambda(y) \wedge \bigwedge_{x \rightarrow y \in H} \neg(\gamma) \top$ otherwise;
5. $\psi \in t^{-1}(\langle \hat{c} \rangle \phi)$ if one of the following holds:
   a. either $\psi \in t^{-1}(\phi)$;
b. or there is an $x_0$ that occurs $\Gamma$-liquid in $t$, and a $\chi \in t^{-1}(\varphi)$ such that $\psi(x) = (\bar{\tau})\chi(x)$ if $x = x_0$, and $\psi(x) = \chi(x)$ otherwise;

c. or there is a $\Gamma$-impatient $P$-rule $H \frac{u}{v}$ and a $\chi \in u^{-1}(\varphi)$ such that $\psi(x) = \chi(x) \land 
\bigwedge_{x \in \gamma \in H} (\bar{\gamma})\chi(y) \land \bigwedge_{x \in \gamma \in H} (\bar{\gamma})\chi(z) \land$ if $x \in \var(t)$, and $\psi(x) = \top$ otherwise;

d. $\psi \in \sigma(t)^{-1}(\varphi)$ for a non-injective substitution $\sigma : \var(t) \to V$ iff there is a $\chi \in t^{-1}(\varphi)$ such that $\psi(x) = \bigwedge_{x \in \sigma^{-1}(x)} \chi(z)$ for all $x \in V$.

The following theorem will be the key to the forthcoming congruence results.

$\blacktriangleright$ **Theorem 10** ([10]). Let $P = (\Sigma, A$, $R)$ be a $\Gamma$-patient complete standard TSS in ready simulation format. For each term $t \in T(\Sigma)$, closed substitution $\rho$, and $\varphi \in \mathcal{O}$:

$$\rho(t) \models \varphi \iff \exists \psi \in t^{-1}(\varphi) \ \forall x \in \var(t) : \rho(x) \models \psi(x).$$

### 3 Stability-respecting branching bisimilarity as a congruence

We proceed to apply the decomposition method from the previous section to derive congruence formats for stability-respecting branching bisimilarity and rooted stability-respecting branching bisimilarity. The idea behind the construction of these congruence formats is that the format must guarantee that a formula from the characterising logic of the equivalence under consideration is always decomposed into formulas from this same logic (see Proposition 14). This implies the desired congruence results (see Theorem 15 and Theorem 16).

The definitions of the congruence formats for (rooted) stability-respecting branching bisimilarity, below, presuppose two predicates $\Lambda$ and $\mathcal{N}$ on the arguments of the function symbols of a TSS: $\Lambda$ marks arguments that contain processes that have started executing, while $\mathcal{N}$ marks arguments that contain processes that can execute immediately. For example, in process algebra, $\Lambda$ and $\mathcal{N}$ typically hold for the arguments of the merge $t_1 \parallel t_2$, and for the first argument of sequential composition $t_1$. $t_2$, and do not hold for the second argument of sequential composition. $\Lambda$ does not hold and $\mathcal{N}$ holds for the arguments of alternative composition $t_1 + t_2$. We will instantiate $\Gamma$ (from Section 2.5) with $\mathcal{N} \cap \Lambda$.

We define when a standard ntytt rule is rooted stability-respecting branching bisimulation safe, and base the rooted stability-respecting branching bisimulation format on that notion. The stability-respecting branching bisimulation format is defined by adding one additional restriction to its rooted counterpart: $\Lambda$ holds for all arguments.

$\blacktriangleright$ **Definition 11.** A standard ntytt rule $\tau = \frac{H}{V} \quad$ is rooted stability-respecting branching bisimulation safe w.r.t. predicates $\mathcal{N}$ and $\Lambda$ if it satisfies the following conditions.

1. Right-hand sides of positive premises occur only $\Lambda$-liquid in $u$.
2. If $x \in \var(t)$ occurs only $\Lambda$-liquid in $t$, then $x$ occurs only $\Lambda$-liquid in $r$.
3. If $x \in \var(t)$ occurs only $\mathcal{N}$-frozen in $t$, then $x$ occurs only $\mathcal{N}$-frozen in $H$.
4. Suppose $x$ has exactly one $\mathcal{N}$-liquid occurrence in $t$, and this occurrence is also $\Lambda$-liquid.
   a. If $x$ has an $\mathcal{N}$-liquid occurrence in a negative premise in $H$ or more than one $\mathcal{N}$-liquid occurrence in the positive premises in $H$, then there is a premise $v \frac{\tau}{\varphi}$ in $H$ such that $x$ occurs $\mathcal{N}$-liquid in $v$.
   b. If there is a premise $w \frac{\tau}{\varphi}$ in $H$ and $x$ occurs $\mathcal{N}$-liquid in $w$, then $r$ is $\mathcal{N} \cap \Lambda$-patient.

Conditions 1–3 were copied from the definition of rooted branching bisimulation safeness from [10], and condition 4b is part of condition 4 in that definition. Condition 4a, however, establishes a relaxation of condition 4 from the definition in [10], where it is required that $x$ has at most one $\mathcal{N}$-liquid occurrence in $H$, which must be in a positive premise. Here, owing to stability, we can be more tolerant, as long as $x \frac{\tau}{\varphi}$ can be derived. As a consequence
of this relaxation the rule for the priority operator is rooted stability-respecting branching bisimulation safe, while it is not rooted branching bisimulation safe.

- **Definition 12.** A standard TSS is in *rooted stability-respecting branching bisimulation format* if it is in ready simulation format and, for some \( R \) and \( \Lambda \), it is \( R\Lambda \)-patient and its rules are all rooted stability-respecting branching bisimulation safe w.r.t. \( R \) and \( \Lambda \). This TSS is in *stability-respecting branching bisimulation format* if moreover \( \Lambda \) is universal.

Since the definition of modal decomposition is based on the \( P \)-ruloids, we must verify that if \( P \) is in rooted stability-respecting branching bisimulation format, then \( P \)-ruloids are rooted stability-respecting branching bisimulation safe.

- **Proposition 13.** Let \( P \) be an \( R\Lambda \)-patient TSS in ready simulation format, in which each rule is rooted stability-respecting branching bisimulation safe w.r.t. \( R \) and \( \Lambda \). Then each \( P \)-ruloid is rooted stability-respecting branching bisimulation safe w.r.t. \( R \) and \( \Lambda \).

Consider a standard TSS in rooted stability-respecting branching bisimulation format, w.r.t. some \( R \) and \( \Lambda \). Definition 9 yields decomposition mappings \( \psi \in t^{-1}(\varphi) \), with \( \Gamma := R\Lambda \). We now prove that if \( \varphi \in O^r_{tb} \), then \( \psi(x) \in O^s_{tb} \) if \( x \) occurs only \( \Lambda \)-liquid in \( t \). (That is why in the stability-respecting branching bisimulation format, \( \Lambda \) must be universal.) Furthermore, we prove that if \( \varphi \in O^r_{tb} \), then \( \psi(x) \in O^s_{tb} \) for all \( x \).

- **Proposition 14.** Let \( P \) be an \( R\Lambda \)-patient standard TSS in ready simulation format, in which each rule is rooted stability-respecting branching bisimulation safe w.r.t. \( R \) and \( \Lambda \).
  1. For each term \( t \) and \( x \) that occurs only \( \Lambda \)-liquid in \( t \): \( \varphi \in O^r_{tb} \Rightarrow \forall \psi \in t^{-1}(\varphi) : \psi(x) \in O^s_{tb} \).
  2. For each term \( t \) and variable \( x \): \( \varphi \in O^r_{tb} \Rightarrow \forall \psi \in t^{-1}(\varphi) : \psi(x) \in O^s_{tb} \).

Now the promised congruence results for \( \Theta^p \) and \( \Theta^r_{tb} \) can be proved in the same way as their counterparts for \( \Theta^p \) and \( \Theta^r_{tb} \) in \([10]\).

- **Theorem 15.** Let \( P \) be a complete standard TSS in stability-respecting branching bisimulation format. Then \( \Theta^p \) is a congruence for \( P \).

- **Theorem 16.** Let \( P \) be a complete standard TSS in rooted stability-respecting branching bisimulation format. Then \( \Theta^r_{tb} \) is a congruence for \( P \).

The *priority* operator \([1]\) is a unary function the definition of which is based on an ordering \( < \) on atomic actions. The term \( \Theta(p) \) executes the transitions of the term \( p \), with the restriction that a transition \( p \xrightarrow{a} q \) only gives rise to a transition \( \Theta(p) \xrightarrow{a} \Theta(q) \) if there does not exist a transition \( p \xrightarrow{b} q' \) with \( b > a \). This intuition is captured by the rule

\[
\Theta(x) \xrightarrow{y} \Theta(y).
\]

In view of the target \( \Theta(y) \), by condition 1 of Definition 11, the argument of \( \Theta \) must be chosen \( \Lambda \)-liquid. And in view of condition 3 of Definition 11, the argument of \( \Theta \) must be \( R \)-liquid. The rule above is rooted stability-respecting branching bisimulation safe, if the following condition on the ordering on atomic actions is satisfied: if \( b > a \), then \( \tau > a \). Namely, this guarantees condition 4a of Definition 11: if there is a premise \( x \xrightarrow{b} \gamma \), then there is also a premise \( x \xrightarrow{\gamma} \).

- **Corollary 17.** \( \Theta^p \) and \( \Theta^r_{tb} \) are congruences for the priority operator.

The priority operator \( \Theta \) does not preserve \( \Theta^r_{tb} \) (cf. \([22, \text{pp. 130–132}]\)). So inevitably, as observed in \([10]\), the rule for \( \Theta \) is not in the rooted branching bisimulation format. Namely, the \( R\Lambda \)-liquid argument \( x \) in the source occurs \( R \)-liquid in the negative premises, which violates the more restrictive condition 4 of the rooted branching bisimulation format.
4 Divergence-preserving branching bisimilarity as a congruence

A modal characterisation of divergence-preserving branching bisimilarity is obtained by adding a unary modality $\Delta$ to the modal logic for branching bisimilarity: $p \models \Delta \varphi$ if there is an infinite trace $p = p_0 \xrightarrow{\alpha} p_1 \xrightarrow{\tau} p_2 \xrightarrow{\beta} \cdots$ such that $p_i \models \varphi$ for all $i \in \mathbb{N}$. Modal formulas $\Delta \varphi$ however elude the inductive decomposition method from Definition 9, because they ask for the existence of an infinite sequence of $\tau$-transitions, as shown by the following example.

$\triangleright$ Example 18. Let $A = \{a_i, b_i \mid i \in \mathbb{N}\} \cup \{c\}$. Parallel composition $\parallel$ is defined by the rules $\frac{x_1 \parallel x_2 \xrightarrow{\alpha} y_1 \parallel y_2}{x_1 \parallel x_2 \xrightarrow{\alpha} y_1 \parallel y_2}$ and $\frac{x_1 \parallel x_2 \xrightarrow{\tau} x_1 \parallel x_2}{x_1 \parallel x_2 \xrightarrow{\tau} x_1 \parallel x_2}$, where $\alpha$ ranges over $A$. We extend its operational semantics with asymmetric communication rules $\frac{x_1 \parallel y_1 \xrightarrow{a_i} x_2 \parallel y_2}{x_1 \parallel x_2 \xrightarrow{\tau} x_1 \parallel x_2}$ for all $i \in \mathbb{N}$. We define $p_i \xrightarrow{a_i} 0$ and $q_i \xrightarrow{\tau} q_{i+1}$ and $q_i \xrightarrow{b_i} 0$ for all $i \in \mathbb{N}$. Then $p_0 \parallel q_0 \models \Delta(\epsilon)(\epsilon \top)$. There is no obvious way to decompose this into modal properties of its arguments $p_0$ and $q_0$.

We circumvent this problem by introducing so-called abstraction-free TSSs that allow only patience rules and rules without premises to carry a conclusion with the label $\tau$, and by introducing oracle transitions $p \xrightarrow{\alpha} \sqrt{\cdot}$, where the transition label $\omega$ reveals some pertinent information on the behaviour of the process $p$, such as whether $p$ can diverge. On abstraction-free TSSs with appropriate oracle transitions the equivalences $\iff_\omega$ and $\iff_{\omega, \Delta}$ coincide; so there our congruence format for $\iff_{\omega, \Delta}$ is also a congruence format for $\iff_{\omega, \Delta}$. We extend this observation to general TSSs by encoding any given TSS into an abstraction-free TSS with oracle transitions, and conclude that on any TSS in the stability-respecting branching bisimulation format $\iff_{\omega, \Delta}$ is a congruence. The same proof strategy shows that on any such TSS also $\iff_{\Delta}^{\top}$ is a congruence, and it extends to the rooted case. In Section 4.1 we present this method in more detail. We refrain from introducing the needed machinery, but state exactly which properties of this machinery we require, and prove our main congruence result based on these properties. Instead of dealing with specific equivalences $\iff_{\Delta}^{\top}$ and $\iff_{\omega, \Delta}$, we work with parametric equivalences $\sim$ and $\approx$ where $\sim$ is finer than $\approx$: they will later be instantiated with $\iff_{\omega, \Delta}$ and $\iff_{\omega, \Delta}^{\top}$. This allows a reuse of our work with $\iff_{\omega, \Delta}^{\top}$ and $\iff_{\omega, \Delta}$ in the roles of $\sim$ and $\approx$, as well as with $\iff_{\omega, \Delta}^{\top}$ and $\iff_{\omega, \Delta}^{\top}$ in the role of $\approx$ and $\approx$.

4.1 A framework for lifting congruence formats to finer equivalences

Our general proof idea is illustrated below, where $P$ is a TSS in a congruence format for $\approx$. We show that also $\sim$ is a congruence on $P$. Consider an operator $f$, for simplicity depicted as unary. Given two closed terms $p$ and $q$ in $P$ with $p \sim q$, we show that $f(p) \sim f(q)$. The picture shows a roundabout trajectory from $p$ to $f(p)$. Imagine a similar trajectory from $q$ to $f(q)$—not depicted but hovering above the page.
First we apply a transformation $\mathcal{AFO}$ on $P$, yielding the abstraction-free TSS with oracle transitions $\mathcal{AFO}(P)$, depicted on the right. For each closed term $p$ of $P$ we introduce a constant $\hat{p}$ in $\mathcal{AFO}(P)$, in such away that $p \sim q$ implies $\hat{p} \sim \hat{q}$. Each $n$-ary operator $f$ of $P$ remains an $n$-ary operator $f$ of $\mathcal{AFO}(P)$. Since $\sim \subseteq \approx \wedge \approx \subseteq \sim \approx$ we have $\hat{p} \approx \hat{q}$. We argue that if $P$ is within our congruence format for $\approx$, then the TSS $\mathcal{AFO}(P)$ is also within this congruence format, and conclude from $\hat{p} \approx \hat{q}$ that $f(\hat{p}) \approx f(\hat{q})$. An important result, deferred to Section 4.3, is that on $\mathcal{AFO}(P)$ the equivalences $\sim$ and $\approx$ coincide. Hence $f(\hat{p}) \sim f(\hat{q})$.

Finally, we decode the processes $f(\hat{p})$ and $f(\hat{q})$, aiming to return to the LTS generated by $P$, but actually ending up in another LTS $K$. Our decoding function $\text{dec}$ exactly undoes the effects of the encoding $\text{enc}$, that sent $p$ to $\hat{p}$, so that $\text{dec}(f(\hat{p}))$ is strongly bisimilar with $f(p)$. A crucial property of the function $\text{dec}$, also deferred to Section 4.3, is that it is compositional for $\sim$, meaning that from $f(\hat{p}) \sim f(\hat{q})$ we may conclude $\text{dec}(f(\hat{p})) \sim \text{dec}(f(\hat{q}))$. By imposing the requirement that $\sim$ contains strong bisimilarity $(\supseteq \approx)$, this implies that $f(p) \sim f(q)$.

We now formalise this proof idea. If $P$ is a complete TSS and $G$ is an LTS disjoint from the LTSs associated with $P$, then, in our formalisation, it will sometimes be convenient to write $P \sqcup G$ for the disjoint union of the LTSs associated with $P$ and $G$.

**Theorem 19.** Let $\sim$ and $\approx$ be behavioural equivalences on LTSs, with $\supseteq \subseteq \sim \subseteq \approx$. Let $\mathcal{F}$ be a congruence format for $\approx$, included in the decent ntyft format, and let $\mathcal{AFO}$ be an operation on standard TSSs, where for each TSS $P = (\Sigma, \text{Act}, R)$ the signature $\hat{\Sigma}$ of $\mathcal{AFO}(P)$ contains $\Sigma$ enriched by a fresh constant $\hat{p}$ for each closed term $p$ in $T(\Sigma)$, such that, for each complete standard TSS $P$ in decent ntyft format:

1. also $\mathcal{AFO}(P)$ is a complete standard TSS,
2. if $P$ is in $\mathcal{F}$-format then so is $\mathcal{AFO}(P)$,
3. $p \sim_P q \Rightarrow \hat{p} \sim_{\mathcal{AFO}(P)} \hat{q}$,
4. $\sim_{\mathcal{AFO}(P)}$ and $\approx_{\mathcal{AFO}(P)}$ coincide, and

there is an LTS $K = (\mathcal{F}_K, \text{Act}_K, \sim_K)$, disjoint from $P$, and a $\text{dec} : T(\hat{\Sigma}) \rightarrow \mathcal{F}_K$ such that:

5. $p \sim_{\mathcal{AFO}(P)} q \Rightarrow \text{dec}(p) \sim_K \text{dec}(q)$, and
6. $\text{dec}(p_1, \ldots, p_n) \equiv_{p \in \mathcal{F}_K} \text{dec}(f(\hat{p}_1, \ldots, \hat{p}_n))$ for any $n$-ary $f \in \Sigma$ and $p_1, \ldots, p_n \in T(\Sigma)$.

Then $\mathcal{F}$ is also a congruence format for $\sim$.

**Proof.** Let $P = (\Sigma, \text{Act}, R)$ be a complete standard TSS in $\mathcal{F}$-format. We will show that $\sim_P$ is a congruence for $P$. So let $f \in \Sigma$ be an $n$-ary function symbol, and let $p_1, q_1 \in T(\Sigma)$ with $p_i \sim_P q_i$ for $i = 1, \ldots, n$. We need to show that $f(p_1, \ldots, p_n) \sim_P f(q_1, \ldots, q_n)$.

By requirements 1 and 2, $\mathcal{AFO}(P)$ is a complete standard TSS in $\mathcal{F}$-format; hence $\approx_{\mathcal{AFO}(P)}$ is a congruence for $\mathcal{AFO}(P)$. By requirement 3 $\hat{p}_i \sim_{\mathcal{AFO}(P)} \hat{q}_i$ for $i = 1, \ldots, n$. By requirement 4 also $\sim_{\mathcal{AFO}(P)}$ is a congruence for $\mathcal{AFO}(P)$. So $f(\hat{p}_1, \ldots, \hat{p}_n) \sim_{\mathcal{AFO}(P)} f(\hat{q}_1, \ldots, \hat{q}_n)$. Hence, by requirement 5, $\text{dec}(f(\hat{p}_1, \ldots, \hat{p}_n)) \sim_K \text{dec}(f(\hat{q}_1, \ldots, \hat{q}_n))$. Therefore, by two applications of requirement 6, and the definition of a behavioural equivalence, $f(p_1, \ldots, p_n) \sim_{P \sqcup K} f(q_1, \ldots, q_n)$ and consequently $f(p_1, \ldots, p_n) \sim_P f(q_1, \ldots, q_n)$. \hfill \IEEEQEDBox

### 4.2 Abstraction-freeness

In this section we introduce the machinery needed for Theorem 19, namely the conversion $\mathcal{AFO}$ on TSSs and the function $\text{dec}$ into the LTS $K$. We also establish requirements 1 and 6 of Theorem 19, leaving 2–5 to the applications of Theorem 19 in Section 4.3 for specific instances of $\sim$ and $\approx$. We here take the set of actions $\text{Act}$ used in Section 4.1 to be $A_\tau$.

Again $\Gamma$ denotes a predicate that marks arguments of function symbols. In [9] we called a standard TSS abstraction-free w.r.t. $\Gamma$ if only its $\Gamma$-patience rules carry the label $\tau$ in their
The LTS that allows to incorporate abstraction-freeness; for step 4 this uses the relaxed definition of abstraction-freeness and each transition appears as rule without premises. The operators \(R\) from \(\approx\) and each non-empty subset \(S\) of positive \(\tau\)-premises of \(r\), a copy of \(r\) in which the labels \(\tau\) in the premises in \(S\) are replaced by \(\iota\);

2. \(R_2\) is obtained from \(R_1\) by replacing, in every rule that has a conclusion with the label \(\tau\) and is not a \(\Gamma\)-patience rule, the \(\tau\)-label in the conclusion by \(\iota\);

3. \(R_3\) is obtained from \(R_2\) by adding the premise \(v\dashv\tau\) to each rule with a premise \(v\vdash\tau\);

4. \(R_4\) is obtained from \(R_3\) by the addition of a rule without premises \(\hat{p} \vdash \hat{q}\) for each transition \(p \xrightarrow{\alpha} q\) \(\text{ws}-\text{provable from}\) \(P\);

5. \(R_5\) is obtained from \(R_4\) by adding a rule \(\hat{p} \xrightarrow{\xi(p)} \sqrt{\gamma}\) for each \(p \in T(\Sigma)\) with \(\xi(p)\) defined;

6. \(\tau\)-patient standard TSS that could be called \(\mathcal{A}\mathcal{F}_{\mathcal{F}}(P)\).

For convenience we consider an auxiliary LTS \(G = (P_G, A_\tau, \rightarrow_G)\) with \(P_G = \{\hat{p} \mid p \in T(\Sigma)\}\) and \(\hat{p} \xrightarrow{\alpha} G \hat{q}\) iff \(P \xrightarrow{ws} p \xrightarrow{\alpha} q\), and an auxiliary LTS \(H = (P_H, A_\tau, O, \rightarrow_H)\) with \(P_H = P_G \cup \{\sqrt{\gamma}\}\) and \(\rightarrow_H := \rightarrow_G \cup \{\hat{p} \xrightarrow{\xi(p)} \sqrt{\gamma} \mid \xi(p)\text{ defined}\}\). The LTS \(G\) is simply a disjoint copy of the LTS generated by \(P\).

**Lemma 21.** If \(p \sim p q\) for some \(p, q \in T(\Sigma)\), then \(\hat{p} \sim_G \hat{q}\).

The LTS \(H\) adds oracle transitions to \(G\). The idea is that \(\xi(p)\) is particular for the \(\sim\)-equivalence class of \(p \in T(\Sigma)\), which on the one hand ensures that \(\hat{p} \sim_H \hat{q}\) iff \(\hat{p} \sim_G \hat{q}\), and on the other hand enforces that \(\sim\) and \(\approx\) coincide on \(H\). Namely, if \(\hat{p} \approx_H \hat{q}\) then the oracle action of \(\hat{p}\) can be matched by \(\hat{q}\) (and vice versa), which implies \(\hat{p} \sim_H \hat{q}\).

Steps 4 and 5 of Definition 20 incorporate the entire LTS \(H\) into \(\mathcal{A}\mathcal{F}_{\mathcal{F}}(P)\): each state appears as a constant and each transition appears as rule without premises. The operators from \(\Sigma\) can now be applied to arguments of the form \(\hat{p}\). Finally, step 6 lets any term \(f(x_1, \ldots, x_n)\) inherit the oracle transitions from its \(\Gamma\)-liquid arguments. Steps 4, 5 and 6 preserve abstraction-freeness; for step 4 this uses the relaxed definition of abstraction-freeness that allows to incorporate \(\tau\)-transitions between constants as rules without premises.

**Example 22.** Let \(P\) have the rules

\[
\begin{align*}
\frac{x_1 \xrightarrow{\tau} y}{g(x_1, x_2, x_3) \xrightarrow{\tau} g(y, x_2, x_3)} & \quad \frac{x_1 \xrightarrow{\alpha} y_1}{g(x_1, x_2, x_3) \xrightarrow{\alpha} x_2} \\
\frac{x_1 \xrightarrow{\tau} y_2}{g(x_1, x_2, x_3) \xrightarrow{\tau} x_2} & \quad \frac{x_2 \xrightarrow{\tau} y}{g(x_1, x_2, x_3) \xrightarrow{\tau} g(y, x_2, x_3)} \\
\frac{x_3 \xrightarrow{\tau} y_1}{g(x_1, x_2, x_3) \xrightarrow{\tau} y_1} & \quad \frac{x_3 \xrightarrow{\tau} y_3}{g(x_1, x_2, x_3) \xrightarrow{\tau} g(y, x_2, x_3)}
\end{align*}
\]

CONCUR 2017
Then the \( \mathcal{AFO}\O\Gamma(P) \) has the rules

\[
\begin{align*}
\frac{x_1 \xrightarrow{r} y}{g(x_1, x_2, x_3) \xrightarrow{r} g(y, x_2, x_3)} & \quad \frac{x_1 \xrightarrow{a} y}{g(x_1, x_2, x_3) \xrightarrow{a} x_2} & \quad \frac{x_1 \xrightarrow{r} y}{g(x_1, x_2, x_3) \xrightarrow{r} y} \\
\frac{x_1 \xrightarrow{a} y}{g(x_1, x_2, x_3) \xrightarrow{a} y} & \quad \frac{x_1 \xrightarrow{r} y}{g(x_1, x_2, x_3) \xrightarrow{r} y} & \quad \frac{x_1 \xrightarrow{a} y}{g(x_1, x_2, x_3) \xrightarrow{a} y} \\
\frac{x_1 \xrightarrow{a} y}{g(x_1, x_2, x_3) \xrightarrow{a} y} & \quad \frac{x_1 \xrightarrow{r} y}{g(x_1, x_2, x_3) \xrightarrow{r} y} & \quad \frac{x_1 \xrightarrow{a} y}{g(x_1, x_2, x_3) \xrightarrow{a} y}
\end{align*}
\]

This illustrates steps 1, 2, 3, 6 of Definition 20. Since there are no closed terms, steps 4, 5 are void.

Clearly, for any \( \Gamma\)-patient standard TSS \( P \), the standard TSS \( \mathcal{AFO}\O\Gamma(P) \) is \( \Gamma \)-patient and abstraction-free w.r.t. \( \Gamma \). We drop superscripts \( O \) and \( \zeta \), and \( \mathcal{AFO}(P) \) denotes \( \mathcal{AFO}\O\Gamma(P) \) for the largest \( \Gamma \) for which \( P \) is \( \Gamma \)-patient. The signature of \( \mathcal{AFO}(P) \) contains the signature of \( P \) enriched by a fresh constant \( \hat{\rho} \) for each closed term \( p \) in \( P \), as required in Theorem 19.

\[ \text{Lemma 23.} \quad \text{Let } P \text{ be a complete standard TSS in ntyft format. If } \hat{\rho} \sim_{\Gamma} \hat{q} \text{ for some } p, q \in T(\Sigma), \text{ then } \hat{\rho} \sim_{\mathcal{AFO}(P)} \hat{q}. \]

As an immediate consequence of Lemmas 21 and 23 we have the following corollary.

\[ \text{Corollary 24.} \quad \text{Requirement 3 of Theorem 19 is met if } \hat{\rho} \sim_{\Gamma} \hat{q} \text{ implies } \hat{\rho} \sim_{\mathcal{AFO}(P)} \hat{q}. \]

The inference \( \hat{\rho} \sim_{\Gamma} \hat{q} \Rightarrow \hat{\rho} \sim_{\mathcal{AFO}(P)} \hat{q} \) depends on the choice of \( O \) and \( \zeta \); it is deferred to Section 4.3.

The oracle inheritance rules in \( \mathcal{AFO}\O\Gamma(P) \) — introduced in step 6 of Definition 20 — ensure that a closed term \( f(p_1, \ldots, p_n) \) has an outgoing \( \omega \)-transition, for \( \omega \in O \), if one of its \( \Gamma \)-liquid arguments \( p_k \) has such a transition. Ultimately, all such oracle transitions stem from \( \hat{\rho} \). Using that in \( \mathcal{AFO}\O\Gamma(P) \) any term \( p \in T(\Sigma) \) can uniquely be written as \( \rho(t) \) with \( t \in T(\Sigma) \) and \( \rho : \text{var}(t) \rightarrow \mathbb{P}_\mathcal{H} \); this fact can be phrased as follows. Let \( t \in T(\Sigma) \) and \( \rho : \text{var}(t) \rightarrow \mathbb{P}_\mathcal{H} \); then \( \mathcal{AFO}\O\Gamma(P) \vdash_{\omega} \rho(t) \xrightarrow{\omega} \) if \( t \) has a \( \Gamma \)-liquid occurrence of a variable \( x \) with \( \rho(x) \xrightarrow{\omega} \). Using this, we now verify requirement 1 of Theorem 19.

\[ \text{Lemma 25.} \quad \text{If a standard TSS } P \text{ in decent ntyft format is complete, then so is } \mathcal{AFO}(P). \]

The LTS \( \mathcal{K} = (\mathbb{P}_\mathcal{K}, A_\tau, O \cup \{\iota\}, \rightarrow_{\mathcal{K}}) \) has as states \( \mathbb{P}_\mathcal{K} = \{ \text{dec}(p) \mid p \in T(\Sigma) \} \), and its transitions are the ones generated by the rules \( \frac{x \xrightarrow{\alpha} y}{\text{dec}(x) \xrightarrow{\text{dec}(\alpha)} \text{dec}(y)} \) and \( \frac{x \xrightarrow{\alpha} y}{\text{dec}(x) \xrightarrow{\text{dec}(\alpha)} \text{dec}(y)} \), where \( \alpha \) ranges over \( A_\tau \). The operator \( \text{dec} : T(\Sigma) \rightarrow \mathbb{P}_\mathcal{K} \) erases all transitions with labels from \( O \) and renames labels \( \iota \) into \( \tau \). All other transitions are preserved.

We end this section by verifying requirement 6 of Theorem 19. Intuitively, the behaviour of a process \( f(p_1, \ldots, p_n) \) in \( P \) is the same as that of \( f(\hat{p}_1, \ldots, \hat{p}_n) \) in \( \mathcal{AFO}(P) \), except that some \( \tau \)-transitions of the former are turned into \( \iota \)-transitions of the latter process, and some oracle transitions may have been added in the latter. Since any rule in \( \mathcal{AFO}(P) \) with a conclusion labelled by \( A_\tau \cup \{\iota\} \) has positive and negative premises with labels from \( A_\tau \cup \{\iota\} \) only, these oracle transitions have no influence on the derivation of any transitions from \( \mathcal{AFO}(P) \) with labels in \( A_\tau \cup \{\iota\} \). The operator \( \text{dec} \) removes all oracle transitions and renames \( \iota \) into \( \tau \), thereby returning the behaviour of \( f(\hat{p}_1, \ldots, \hat{p}_n) \) to match that of \( f(p_1, \ldots, p_n) \) exactly.

\[ \text{Proposition 26.} \quad \text{Let } P \text{ be a complete standard TSS in decent ntyft format. Then } f(p_1, \ldots, p_n) \sim_{\mathcal{K} \cdot \mathbb{P}_\mathcal{K}} \text{dec}(f(\hat{p}_1, \ldots, \hat{p}_n)) \text{ for any n-ary } f \in \Sigma \text{ and } p_1, \ldots, p_n \in T(\Sigma). \]
4.3 Application of the framework to divergence-preserving semantics

We apply Theorem 19 to show that the stability-respecting branching bisimulation format and its rooted variant are congruence formats for \( \cong^\Delta_b \) and \( \cong^\Delta \), and for \( \cong^\Delta_{rb} \) and \( \cong^\Delta_{rb} \).

As congruence format in Theorem 19 we take the (rooted) stability-respecting branching bisimulation format intersected with the decent ntyft format. It is straightforward to check that the conversion AFO on standard TSSs defined in Section 4.2 preserves the (rooted) stability-respecting branching bisimulation format, and thus satisfies requirement 2 of Theorem 19. Here it is important that in step 6 of Definition 20 a term \( f(x_1, \ldots, x_n) \) inherits oracle transitions only from its \( \Gamma \)-liquid arguments – else condition 3 of Definition 11 would be violated. Furthermore, condition 4a of Definition 11 is preserved because premises \( v \xrightarrow{\tau} \) are kept in place in step 3 of Definition 20; and condition 4b of Definition 11 is preserved because the transformation in Definition 20 does not introduce new positive \( \tau \)-premises.

We first apply Theorem 19 with \( \cong^\Delta_b \) and \( \cong^\Delta_b \) in the roles of \( \sim \) and \( \approx \). Let us say that a process \( p \) in an LTS is divergent if there exists an infinite sequence of processes \( (p_k)_{k \in \mathbb{N}} \) such that \( p_k \xrightarrow{\tau} p_{k+1} \) for all \( k \in \mathbb{N} \), i.e. if \( p \models \Delta \). In the construction of the LTS \( H \) out of \( G \) (cf. Section 4.2) we take \( O = \{ \Delta \} \) and let \( \zeta(g) = \Delta \) iff \( g \) is divergent. Thus in \( H \) all divergent states of \( G \) have a fresh outgoing transition labelled \( \Delta \).

With this definition of \( \sim \), we have \( \hat{p} \sim_H \hat{q} \) iff \( \hat{p} \sim_G \hat{q} \): any weakly divergence-preserving branching bisimulation \( B \) on \( G \) relates divergent states with divergent states only, and thus is also a weakly divergence-preserving branching bisimulation on \( H \) (adding \( \sqrt{\bigcup \sqrt{\bigcup} \sqrt{\bigcup} \sqrt{\bigcup}} \)). Hence, by Corollary 24, requirement 3 of Theorem 19 is satisfied. Requirement 4 is also satisfied:

\textbf{Proposition 27.} On AFO\((P)\) the equivalences \( \equiv^\Delta_b \) and \( \equiv^\Delta_b \) coincide.

The next example shows that Proposition 27 would not hold if we had skipped step 6 of Definition 20, inheriting oracle transitions for \( \Gamma \)-liquid arguments, or had not used oracle transitions at all.

\textbf{Example 28.} Let \( p \in T(\Sigma) \) have no outgoing transitions, while \( q \in T(\Sigma) \) has only a \( \tau \)-transition to itself and a \( \tau \)-transition to \( p \). Then in the LTS \( G \) we have \( \hat{p} \equiv^\Delta_b G \hat{q} \) but \( \hat{p} \not\equiv^\Delta_{rb} G \hat{q} \). After translation to \( H \), the processes \( \hat{p} \) and \( \hat{q} \) are distinguished by means of oracle transitions, so that we have \( \hat{p} \equiv^\Delta_b H \hat{q} \) and \( \hat{p} \not\equiv^\Delta_{rb} H \hat{q} \). Now let \( \Sigma \) feature a unary operator \( f \) with as only rule \( \frac{\rightarrow}{f(x) \xrightarrow{\tau} f(y)} \). If oracle transitions would not be inherited in \( \text{AFO}(P) \), then we would have \( f(\hat{p}) \equiv^\Delta_b \text{AFO}(P) f(\hat{q}) \) but \( f(\hat{p}) \not\equiv^\Delta_{rb} \text{AFO}(P) f(\hat{q}) \).

Also requirement 5 of Theorem 19 holds.

\textbf{Proposition 29.} \( p \equiv^\Delta_{rb} \text{AFO}(P) q \Rightarrow \text{dec}(p) \equiv^\Delta_{rb} \text{dec}(q) \).

\textbf{Corollary 30.} The stability-respecting branching bisimulation format intersected with the decent ntyft format is a congruence format for \( \equiv^\Delta_b \).

Each standard TSS \( P \) in ready simulation format can be converted to a TSS \( P' \) in decent ntyft format, preserving the set of \( ws \)-provable closed literals \([3]\). Moreover, if \( P \) is in stability-respecting branching bisimulation format, then so is \( P' \). Thus we obtain:

\textbf{Theorem 31.} Let \( P \) be a complete standard TSS in stability-respecting branching bisimulation format. Then \( \equiv^\Delta_b \) is a congruence for \( P \).

In a similar fashion it can be proved that the stability-respecting branching bisimulation format is a congruence format for \( \equiv^\Delta_b \), and that the rooted stability-respecting branching bisimulation format is a congruence format for \( \equiv^\Delta_{rb} \) as well as \( \equiv^\Delta_{rb} \).
5 Related work

Ulidowski [19, 20, 21] proposed congruence formats, inside GSOS [4], for weak semantics that take into account non-divergence, called convergence in [11]. In [19] he introduces the ISOS format, and shows that the weak convergent refusal simulation preorder is a precongruence for all TSSs in the ISOS format. The GSOS format – in our terminology the decent ntyxt format – allows only decent ntyft rules with variables as the left-hand sides of premises. The ISOS format is contained in the intersection of the GSOS format and our stability-preserving branching bisimulation format. Its additional restriction is that no variable may occur multiple times as the left-hand side of a positive premise, or both as the left-hand side of a positive premise and in the conclusion of a rule. In [20, 21] he employs Ordered SOS (OSOS) TSSs [20]. An OSOS TSS allows no negative premises, but includes priorities between rules: \( r < r' \) means that \( r \) can only be applied if \( r' \) cannot. An OSOS specification can be seen as, or translated into, a GSOS specification with negative premises. Each rule \( r \) with exactly one higher-priority rule \( r' > r \) is replaced by a number of rules, one for each (positive) premise of \( r' \); in the copy of \( r \), this premise is negated. For a rule \( r \) with multiple higher-priority rules \( r' \), this replacement is carried out for each such \( r' \).

The ebo and bbo formats from [20] target convergent delay and branching bisimulation equivalence, respectively, whereas the rebo and rbbo formats from [21] target their rooted counterparts. These rooted formats are more liberal than their unrooted counterparts, and the (r)bbo format is more liberal than the (r)ebo format. If patience rules are not allowed to have a lower priority than other rules, then the (r)bbo format, upon translation from OSOS to GSOS, can be seen as a subformat of our (rooted) stability-respecting branching bisimulation format. Patience rules are in the (r)bbo format however, under strict conditions, allowed to be dominated by other rules, which in our setting gives rise to patience rules with negative premises. This is outside the realm of our rooted stability-respecting branching bisimulation format. On the other hand, the TSSs of the process algebra BPA\(_{\subseteq,\tau}\), the binary Kleene star and deadlock testing (see [7]), for which rooted convergent branching bisimulation equivalence is a congruence, are outside rbbo but within the rooted stability-respecting branching bisimulation format.

6 Conclusions

We showed how the method from [10] for deriving congruence formats through modal decomposition can be applied to weak semantics that are stability-respecting. We used (rooted and unrooted) stability-respecting branching bisimulation equivalence as a notable example. Moreover, we developed a general method for lifting congruence formats from a weak semantics to a finer semantics, and used it to show that congruence formats for \( \equiv_r^s \) and \( \equiv_b^s \) are also congruence formats for their divergence-preserving counterparts. This research provides a deeper insight into the link between modal logic and congruence formats, and strengthens the framework from [10] for the derivation of congruence formats for weak semantics.

We build on a rich body of earlier work in the realm of structural operational semantics: the notions of well-supported proofs and complete TSSs from [12]; the ntyft/ntyxt format [16, 5]; the transformation to ruloids; and the work on modal decomposition and congruence formats from [3]. In spite of these technicalities, the resulting framework for deriving congruence formats for weak semantics is relatively straightforward. For this one only needs to: (1) provide a modal characterisation of the weak semantics under consideration; (2) study the class of modal formulas that result from decomposing this modal characterisation, and formulate syntactic restrictions on TSSs to bring this class of modal formulas within the
original modal characterisation; and (3) check that these syntactic restrictions are preserved under the transformation to ruloids. Steps (2) and (3) are very similar in structure for different weak semantics, as exemplified by the way we obtained a congruence format for stability-respecting branching bisimulation equivalence. And the resulting congruence formats tend to be more liberal and elegant than existing congruence formats in the literature.

Our intention is to carve out congruence formats for all weak semantics in the spectrum from [11] that have reasonable congruence properties. At first we expected that the current third instalment would allow us to do so. However, it turns out that convergent weak semantics as considered in for instance [20, 21, 23] still need extra work. The modal characterisations of these semantics are three-valued [11], which requires an extension of the modal decomposition technique to a three-valued setting.

References


Checking Linearizability of Concurrent Priority Queues

Ahmed Bouajjani\textsuperscript{1}, Constantin Enea\textsuperscript{2}, and Chao Wang\textsuperscript{3}

\textsuperscript{1} IRIF, University Paris Diderot, Paris, France
about@irif.fr
\textsuperscript{2} IRIF, University Paris Diderot, Paris, France
cenea@irif.fr
\textsuperscript{3} IRIF, University Paris Diderot, Paris, France
wangch@irif.fr

Abstract

Efficient implementations of concurrent objects such as atomic collections are essential to modern computing. Unfortunately their correctness criteria — linearizability with respect to given ADT specifications — are hard to verify. Verifying linearizability is undecidable in general, even on classes of implementations where the usual control-state reachability is decidable. In this work we consider concurrent priority queues which are fundamental to many multi-threaded applications like task scheduling or discrete event simulation, and show that verifying linearizability of such implementations is reducible to control-state reachability. This reduction entails the first decidability results for verifying concurrent priority queues with an unbounded number of threads, and it enables the application of existing safety-verification tools for establishing their correctness.

1998 ACM Subject Classification D.2.4 Software/Program Verification

Keywords and phrases Concurrency, Linearizability, Model Checking

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2017.16

1 Introduction

Multithreaded software is typically built with specialized “concurrent objects” like atomic integers, queues, maps, priority queues. These objects’ methods are designed to conform to better established sequential specifications, a property known as linearizability [14], despite being optimized to avoid blocking and exploit parallelism, e.g., by using machine instructions like compare-and-swap. Intuitively, linearizability asks that every individual operation appears to take place instantaneously at some point between its invocation and its return. Verifying linearizability is intrinsically hard, and undecidable in general [4]. However, recent work [5] has shown that for particular objects, e.g., registers, mutexes, queues, and stacks, the problem of verifying linearizability becomes decidable (for finite-state implementations).

In this paper, we consider another important object, namely the priority queue, which is essential for applications such as task scheduling and discrete event simulation. Numerous implementations have been proposed in the research literature, e.g., [2, 8, 16, 20, 19], and concrete implementations exist in many modern languages like C++ or Java. Priority queues are collections providing put and rm methods for adding and removing values. Every added value is associated to a priority and a remove operation returns a minimal priority value.
For generality, we consider a partially-ordered set of priorities. Values with incomparable priorities can be removed in any order, and values having the same priority are removed in the FIFO order. Implementations like the PriorityBlockingQueue in Java where same priority values are removed in an arbitrary order can be modeled in our framework by renaming equal priorities to incomparable priorities (while preserving the order constraints).

Compared to previously studied collections like stacks and queues, the main challenge in dealing with priority queues is that the order in which values are removed is not fixed by the happens-before between add/remove operations (e.g., in the case of queues, values are removed in the order in which they were inserted), but by parameters of the put operations (the priorities) which come from an unbounded domain. For instance, the sequential behavior $\text{put}(a, p_1) \cdot \text{put}(b, p_3) \cdot \text{put}(c, p_2) \cdot \text{rm}(a, p_1) \cdot \text{rm}(c, p_2)$ where the priority $p_1$ is less than $p_2$ which is less than $p_3$, is not admitted neither by the regular queue nor the stack.

We give a characterization of concurrent priority queue behaviors violating linearizability in terms of automata. This characterization enables a reduction of checking linearizability for arbitrary implementations to reachability or invariant checking, and implies decidability for checking linearizability of finite-state implementations. While linearizability violations for stacks and queues can be described using finite-state automata [5], the case of priority queues requires register automata where registers are used to store and compare priorities.

This characterization is obtained in several steps. We define a recursive procedure that recognizes valid sequential executions, which is then extended to recognize linearizable concurrent executions. Intuitively, for an execution $e$, this procedure deals with values occurring in $e$ one by one, starting with values of maximal priority (to be removed the latest). For each value $x$, it checks whether $e$ satisfies some property “local” to that value, i.e., which is agnostic to how the operations adding or removing other values are ordered between them (w.r.t. the happens-before), other than how they are ordered w.r.t. the operations on $x$. When this property holds, the procedure is applied recursively on the rest of the execution, without the operations on $x$. This procedure works only for executions where a value is added at most once, but this is not a limitation for data-independent implementations whose behavior doesn’t depend on the values that are added or removed. In fact, all the implementations that we are aware of are data-independent.

Next, we show that checking whether an execution violates this “local” property for a value $x$ can be done using a class of register automata [15, 9, 18] (transition systems where the states consist of a fixed set of registers that can receive values and be compared). Actually, only two registers are needed: one register $r_1$ for storing a priority guessed at the initial state, and one register $r_2$ for reading priorities as they occur in the execution and comparing them with the one stored in $r_1$. We show that registers storing values added to or removed from the priority queue are not needed, since any data-independent implementation admits a violation to linearizability whenever it admits a violation where the number of values is constant, and at most 4 (the number of priorities can still be unbounded).

The remainder of this article is organized as follows. Section 2 describes the priority queue ADT, lists several semantic properties like data-independence, and recalls the notion of linearizability. Section 3 defines a recursive procedure for checking linearizability of concurrent priority queue behaviors. Section 4 gives an automata characterization of the violations to linearizability, and Section 5 discusses related work. Detailed proofs and constructions can be found in the extended version [7].
2 The Priority Queue ADT

We consider priority queues whose interface contains two methods put and rm for adding and respectively, removing a value. Each value is assigned with a priority when being added to the data structure (by calling put) and the remove method rm removes a value with a minimal priority. For generality, we assume that the set of priorities is partially-ordered. Incomparable priorities can be removed in any order. When multiple values are assigned with the same priority, rm returns the least recent value. Also, when the set of values stored in the priority queue is empty, rm returns the distinguished value empty. In this section, we formalize (concurrent) executions and implementations, introduce a set of properties satisfied by all the implementations we are aware of, and recall the standard correctness criterion for concurrent implementations of ADTs known as linearizability [14].

2.1 Executions

We fix a (possibly infinite) set D of data values, a (possibly infinite) set P of priorities, a partial order $\prec$ among elements in P, and an infinite set O of operation identifiers. The latter are used to match call and return actions of the same invocation. Call actions $call_o(put,a,p)$ and $call_o(rm,a')$ with $a \in D$, $a' \in D \cup \{empty\}$, $p \in P$, and $o \in O$, combine a method name and a set of arguments with an operation identifier. The return value of a remove is transformed to an argument value for uniformity. The return actions are denoted in a similar way as $ret_o(put,a,p)$ and respectively, $ret_o(rm,a')$.

An execution $e$ is a sequence of call and return actions which satisfy the following well-formedness properties: each return is preceded by a matching call (having the same operation identifier), and each operation identifier is used in at most one call/return. We assume every set of executions is closed under isomorphic renaming of operation identifiers. An $m(a)$-operation in an execution $e$ is an operation identifier $o$ s.t. $e$ contains the actions $call_o(m,a)$ and $ret_o(m,a)$. An execution is called sequential when no two operations overlap, i.e., each call action is immediately followed by its matching return action, and concurrent otherwise. For readability, we write a sequential execution as a sequence of put(a, p) and rm(a) symbols representing a pair of actions $call_o(put,a,p) \cdot ret_o(put,a,p)$ and $call_o(rm,a) \cdot ret_o(rm,a)$, respectively ($o \in O$). For example, given two priorities $p_1 \prec p_2$, put(a, p2) · put(b, p1) · rm(b) is a sequential execution of the priority queue (rm returns b because it has smaller priority).

We define SeqPQ, the set of sequential priority queue executions, semantically via a labelled transition system (LTS, for short). An LTS is a tuple $A = (Q, \Sigma, \rightarrow, q_0)$, where $Q$ is a set of states, $\Sigma$ is an alphabet of transition labels, $\rightarrow \subseteq Q \times \Sigma \times Q$ is a transition relation, and $q_0$ is the initial state. We model the priority queue as an LTS PQ where states are mappings associating priorities in P with sequences of values in D, representing a snapshot of the priority queue (for each priority, the values are ordered as they were inserted), and the transition labels are $put(a,p)$ and $rm(a)$. Each transition modifies the state as expected.

For example, $q_1 \xrightarrow{rm(empty)} q_2$ if $q_1 = q_2$, and $q_1$ and $q_2$ map each priority to the empty sequence $e$. Then, SeqPQ is the set of traces (words) accepted by PQ.

An implementation $I$ is a set of executions. Implementations represent libraries whose methods are called by external programs. In the remainder of this work, we consider only

---

1 Method return values are guessed nondeterministically, and validated at return points. This can be handled using assume statements, which only admit executions satisfying a given predicate.
completed executions, where each call action has a corresponding return action. This simplification is sound when the method invocations can always make progress in isolation.

2.2 Semantic Properties of Priority Queues

We define two properties which are important for our results: (1) data independence [22, 1] states that priority queue behaviors do not depend on the actual values which are added to the queue, and (2) closure under projection [5] states that executions remain valid by removing all the operations adding or removing certain values.

An execution \( e \) is data-differentiated if every value is added at most once, i.e., for each \( d \in \mathcal{D} \), \( e \) contains at most one action \( \text{call}_o(\text{put}, d, p) \) with \( o \in \mathcal{O} \) and \( p \in \mathcal{P} \). Note that this property concerns only values, a data-differentiated execution \( e \) may contain more than one value with the same priority. The subset of data-differentiated executions of a set of executions \( E \) is denoted by \( E_e \).

A renaming function \( r \) is a function from \( \mathcal{D} \) to \( \mathcal{D} \). Given an execution \( e \), we denote by \( r(e) \) the execution obtained from \( e \) by replacing every data value \( x \) by \( r(x) \). Note that \( r \) renames only the values and keeps the priorities unchanged. Intuitively, renaming values has no influence on the behavior of the priority queue, contrary to renaming priorities.

**Definition 1.** A set of executions \( E \) is data independent iff

- for all \( e \in E \), there exists \( e' \in E_e \) and a renaming function \( r \), such that \( e = r(e') \),
- for all \( e \in E \) and for all renamings \( r \), \( r(e) \in E \).

The following lemma is a direct consequence of definitions.

**Lemma 2.** \( \text{SeqPQ} \) is data independent.

Beyond sequential executions, every concurrent priority queue implementation that we are aware of is data-independent. From now on, we consider only data-independent implementations. This assumption enables a reduction from checking the correctness of an implementation \( I \) to checking the correctness of its data-differentiated executions in \( I_e \).

Besides data independence, the sequential executions of the priority queue satisfy the following closure property: an execution remains valid when removing all the operations with an argument in some set of values \( D \subseteq \mathcal{D} \) and any \( \text{rm}(\text{empty}) \) operation (since they are read-only and they don’t affect the queue’s state). To distinguish between different \( \text{rm}(\text{empty}) \) operations while simplifying the technical exposition, we assume that they receive as argument a value, i.e., call actions are of the form \( \text{call}_o(\text{rm}, \text{empty}, a) \) for some \( a \in \mathcal{D} \). We will make explicit this argument only when needed in our technical development. The projection \( e|D \) of an execution \( e \) to a set of values \( D \subseteq \mathcal{D} \) is obtained from \( e \) by erasing all the call/return actions with an argument not in \( D \). We write \( e \setminus x \) for the projection \( e|_{\mathcal{D} \setminus \{x\}} \).

Let \( \text{proj}(e) \) be the set of all projections of \( e \) to a set of values \( D \subseteq \mathcal{D} \).

**Lemma 3.** \( \text{SeqPQ} \) is closed under projection, i.e., \( \text{proj}(e) \subseteq \text{SeqPQ} \) for each \( e \in \text{SeqPQ} \).

2.3 Linearizability

We recall the notion of linearizability [14] which is the de facto standard correctness condition for concurrent data structures. Given an execution \( e \), the happen-before relation \( <_{hb} \) between operations \( o_1 <_{hb} o_2 \) is defined as follows: \( o_1 <_{hb} o_2 \), if the return action of \( o_1 \) occurs before

\[<_{hb}\]
the call action of $o_2$ in $e$. The happens-before relation is an interval order [6]: for distinct $o_1, o_2, o_3, o_4$, if $o_1 <_{hb} o_2$ and $o_3 <_{hb} o_4$, then either $o_1 <_{hb} o_4$, or $o_3 <_{hb} o_2$. Intuitively, this comes from the fact that concurrent threads share a notion of global time.

Given a (concurrent) execution $e$ and a sequential execution $s$, we say that $e$ is linearizable w.r.t $s$, denoted $e \sqsubseteq s$, if there is a bijection $f : O_1 \rightarrow O_2$, where $O_1$ and $O_2$ are the set of operations of $e$ and $s$, respectively, such that (1) the call and return actions with identifier $o$ and $f(o)$, respectively, are the same and (2) if $o_1 <_{hb} o_2$, then $f(o_1) <_{hb} f(o_2)$. A (concurrent) execution $e$ is linearizable w.r.t. a set $S$ of sequential executions, denoted $e \sqsubseteq S$, if there exists $s \in S$ such that $e \sqsubseteq s$. A set of concurrent executions $E$ is linearizable w.r.t. $S$, denoted $E \sqsubseteq S$, if $e \sqsubseteq S$ for all $e \in E$.

The following lemma states that by data-independence, it is enough to consider only data-differentiated executions when checking linearizability. Section 3 will focus on characterizing linearizability for data-differentiated executions.

**Lemma 4.** A data-independent implementation $I$ is linearizable w.r.t. a data-independent set $S$ of sequential executions, if and only if $I \not\sqsubseteq S$ is linearizable w.r.t. $S$.

### 3 Checking Linearizability of Priority Queue Executions

We define a recursive procedure for checking linearizability of a data-differentiated execution w.r.t. $\text{SeqPQ}$. To ease the exposition, Section 3.1 introduces a recursive procedure for checking whether a data-differentiated sequential execution is admitted by the priority queue which is then extended to the concurrent case in Section 3.2.

#### 3.1 Characterizing Data-Differentiated Sequential Executions

The recursive procedure $\text{Check-PQ-Seq}$ outlined in Algorithm 1 checks whether a data-differentiated sequential execution belongs to $\text{SeqPQ}$ (i.e., if it is accepted by the LTS $PQ$). Roughly, it selects one or two operations in the input execution, checks whether their return values are correct by ignoring the order between the other operations other than how they are ordered w.r.t. the selected ones, and calls itself recursively on the execution without the selected operations.

We explain how the procedure works on the following execution:

$$\text{put}(c, p_2) \cdot \text{put}(a, p_1) \cdot \text{rm}(a) \cdot \text{rm}(\text{empty}) \cdot \text{put}(d, p_2) \cdot \text{put}(f, p_3) \cdot \text{rm}(f) \cdot \text{put}(b, p_1) \tag{1}$$

where $p_1, p_2, p_3$ are priorities such that $p_1 < p_2$ and $p_1 < p_3$, and $p_2$ and $p_3$ are incomparable. Since the $\text{rm}(\text{empty})$ operations are read-only (they don’t affect the queue’s state), they are selected first. An $\text{rm}(\text{empty})$-operation $o$ is correct when every $\text{put}(x, p)$ operation before $o$ is matched to a $\text{rm}(x)$ operation which also occurs before $o$. This is true in this case for $x \in \{a, c\}$. Thus, the correctness of (1) reduces to the correctness of

$$\text{put}(c, p_2) \cdot \text{put}(a, p_1) \cdot \text{rm}(a) \cdot \text{rm}(\text{empty}) \cdot \text{put}(d, p_2) \cdot \text{put}(f, p_3) \cdot \text{rm}(f) \cdot \text{put}(b, p_1) \tag{2}$$

When the execution contains no $\text{rm}(\text{empty})$-operation, the procedure selects a $\text{put}$ operation adding a value that is not removed and that has a maximal priority. For (2), it selects $\text{put}(d, p_2)$ because $p_2$ is a maximal priority. This operation is correct since $d$ is the last value with priority $p_2$ in the execution, and the correctness of (2) reduces to the correctness of

$$\text{put}(c, p_2) \cdot \text{put}(a, p_1) \cdot \text{rm}(a) \cdot \text{rm}(\text{empty}) \cdot \text{put}(f, p_3) \cdot \text{rm}(f) \cdot \text{put}(b, p_1) \tag{3}$$
Checking Linearizability of Concurrent Priority Queues

Algorithm 1: Check-PQ-Seq

Input: A data-differentiated sequential execution $e$
Output: true iff $e \in \text{SeqPQ}$

1. if $e = \epsilon$ then
2. return true;
3. if Has-EmptyRemoves($e$) then
4. if $\exists o = \text{rm}(\text{empty}) \in e$ such that $\text{EmptyRemove-Seq}(e, o)$ holds then
5. return Check-PQ-Seq($e \setminus o$);
6. else if Has-UnmatchedMaxPriority($e$) then
7. if $\exists x \in \text{values}(e)$ such that $\text{UnmatchedMaxPriority-Seq}(e, x)$ holds then
8. return Check-PQ-Seq($e \setminus x$);
9. else
10. if $\exists x \in \text{values}(e)$ such that $\text{MatchedMaxPriority-Seq}(e, x)$ holds then
11. return Check-PQ-Seq($e \setminus x$);
12. else
13. return false;

If no operations like above can be found, Check-PQ-Seq selects a pair of put and rm operations adding and removing the same maximal priority value. For (2), it can select $\text{rm}(c, p_2)$ and $\text{rm}(c)$. The value returned by $\text{rm}(c)$ is correct if all the values of priority smaller than $p_2$ added before $\text{rm}(c)$ are also removed before $\text{rm}(c)$. In this case, $a$ is the only value of priority smaller than $p_2$ and it satisfies this property. Applying a similar reasoning for all the remaining values, it can be proved that this execution is correct.

Formally, the selected operations depend on the following set of predicates on executions:

- Has-EmptyRemoves($e$) = true iff $e$ contains a $\text{rm}(\text{empty})$-operation
- Has-UnmatchedMaxPriority($e$) = true iff $p \in \text{unmatched-priorities}(e)$ for a maximal $p$

where $\text{priorities}(e)$, resp., $\text{unmatched-priorities}(e)$, is the set of priorities occurring in put operations of $e$, resp., in put operations of $e$ for which there is no rm operation removing the same value. We call the latter unmatched put operations. A put operation which is not unmatched is called matched. For simplicity, we consider the following syntactic sugar

- Has-MatchedMaxPriority($e$) = $\neg$Has-EmptyRemoves($e$) $\wedge$ $\neg$Has-UnmatchedMaxPriority($e$).

By an abuse of notation, we assume Has-UnmatchedMaxPriority($e$) $\Rightarrow$ $\neg$Has-EmptyRemoves($e$) (this is sound by the order of the conditions in Check-PQ-Seq).

The predicates defining the correctness of the selected operations are defined as follows:

- $\text{EmptyRemove-Seq}(e, o) = \text{true iff } e = u \cdot o \cdot v$ and matched($u$)
- $\text{UnmatchedMaxPriority-Seq}(e, x) = \text{true iff } e = u \cdot \text{put}(x, p) \cdot v$, $p \not\in \text{priorities}(u \cdot v)$,
  $p \not\in \text{unmatched-priorities}(v)$
- $\text{MatchedMaxPriority-Seq}(e, x) = \text{true iff } e = u \cdot \text{put}(x, p) \cdot v \cdot \text{rm}(x) \cdot w$, matched$_\prec$(u, v, p),
  $p \not\in \text{unmatched-priorities}(u \cdot v \cdot w)$, $p \not\in \text{priorities}(u \cdot v \cdot w)$,
  and $p \not\in \text{priorities}(v \cdot w)$

where $p \prec \text{priorities}(e)$ when $p \prec p'$ for some $p' \in \text{priorities}(e)$ (and similarly for $p \prec \text{unmatched-priorities}(e)$ or $p \preceq \text{unmatched-priorities}(e)$), matched$_\prec$(e, p) holds when each value with priority strictly smaller than $p$ is removed in $e$, and matched(e) holds when matched$_\prec$(e, p) holds for each $p \in \mathbb{P}$. Compared to the example presented at the beginning of the section, these predicates take into consideration that multiple values with the same priority are removed in FIFO order: the predicate MatchedMaxPrioritySeq(e, x) holds when $x$ is the last value with priority $p$ added in $e$.

When $o$ is an $\text{rm}(\text{empty})$-operation, $e \setminus o$ is the maximal subsequence of $e$ which doesn’t contain $o$. For an execution $e$, values($e$) is the set of values in call/return actions of $e$. 
The following lemma states the correctness of Check-PQ-Seq.

Lemma 5. Check-PQ-Seq(e) = true iff e ∈ SeqPQ, for every data-differentiated sequential execution e.

3.2 Checking Linearizability of Data-Differentiated Concurrent Executions

The extension of Check-PQ-Seq to concurrent executions, checking whether they are linearizable w.r.t. SeqPQ, is obtained by replacing every predicate Γ-Seq with

\[ Γ-\text{Seq}(e, a) = \text{true} \text{ iff there is a sequential execution } s \text{ such that } e \subseteq s \text{ and } Γ-\text{Seq}(s, a) \]

for each \( Γ \in \{\text{EmptyRemove, UnmatchedMaxPriority, MatchedMaxPriority}\} \). The obtained procedure is denoted by Check-PQ-Conc (recursive calls are modified accordingly).

The following lemma states the correctness of Check-PQ-Conc. Completeness follows easily from the properties of SeqPQ. If Check-PQ-Conc(e) = false, then there exists a set \( D \) of values such that either EmptyRemove-Conc(e,D) is false, or UnmatchedMaxPriority-Conc(e,D,x) is false for all the values \( x \) of maximal priority that are not removed (and there exists at least one such value), or MatchedMaxPriority-Conc(e,D,x) is false for all the values \( x \) of maximal priority (and these values are all removed in e \( \cup D \)). It can be easily seen that we get e \( \cup D \not\subseteq \text{SeqPQ} \) in all cases, which by the closure under projection of SeqPQ implies, e \( \not\subseteq \text{SeqPQ} \) (since every linearization of e includes as a subsequence a linearization of e \( \cup D \)).

Lemma 6. Check-PQ-Conc(e) = true iff e \( \subseteq \text{SeqPQ} \), for every data-differentiated e.

Proving soundness is highly non-trivial and one of the main technical contributions of this paper. The main technical difficulty is showing that for any execution e, any linearization of e \( \setminus x \) for some maximal priority value x can be extended to a linearization of e provided UnmatchedMaxPriority or MatchedMaxPriority holds (depending on whether there are values with the same priority as x in e which are not removed).

We explain the proof of this property on the execution e in Figure 1(a) where \( p_1 \prec p, p_1 \prec p_2 \), and the predicate Has-MatchedMaxPriority(e) holds. Assume that there exist two sequential executions l and \( l' \) such that e \( \subseteq l = u \cdot \text{put}(x, p) \cdot v \cdot \text{rm}(x) \cdot w \), MatchedMaxPriority-Seq(l,x) holds, and e \( \setminus x \subseteq l' \subseteq \text{SeqPQ} \). Let u = e, w be any sequence formed of \( \text{put}(z_2, p_2) \) and \( \text{rm}(z_1) \) (we distinguish them by adding the suffix “-w” to their name, e.g., rm(z_1) - w), and v be any sequence containing the remaining operations. In general, the linearization \( l' \) can be defined by choosing for each operation, a point in time between its call and return, called linearization point. The order between the linearization points defines the sequence \( l' \). Figure 1(a) draws linearization points for the operations in e \( \setminus x \) which define \( l' \). We show how to construct a sequence \( l'' = l_1'' \cdot \text{put}(x, p) \cdot l_2'' \cdot \text{rm}(x) \cdot l_3'' \in \text{SeqPQ} \) s.t. e \( \subseteq l'' \).

An operation is called p-comparable (resp., p-incomparable) when it receives as argument a value of priority comparable to p (resp., incomparable to p). Defining \( l_1'' \), \( l_2'' \), and \( l_3'' \) as the projection of \( l' \) to the set of operations in u, v, and w, respectively, leads to a sequence \( l'' \notin \text{SeqPQ} \). This is because MatchedMaxPriority-Seq(l,x) imposes no restriction on p-incomparable operations in u \( \cdot v \), and the projection of \( l' \) to p-incomparable operations in u \( \cdot v \) is not in SeqPQ. In this example, this projection is \( \text{put}(z_1, p_2) \cdot \text{rm}(z_2) \).

\footnotesize
\(^3\) In general, there may exist multiple ways of choosing linearization points to define the same linearization. Our construction is agnostic to this choice.
We define the sets of operations $U'$, $V'$ and $W'$ such that $l_i'$, $l_j'$ and $l_3'$ are the projections of $l_i$ to $U'$, $V'$, and $W'$, respectively. This is done in two steps:

1. The first step is to define $W'$. The $p$-comparable operations in $W'$ are the same as in $w$. To identify the $p$-incomparable operations in $W'$, we search for a $p$-incomparable operation $o$ which either happens before some $p$-comparable operation in $w$, or whose linearization point occurs after $\text{ret}(rm, x)$. We add to $W'$ the operation $o$ and all the $p$-incomparable operations occurring after $o$ in $l'$. In this example, $o$ is $rm(z_1)$ and the only $p$-incomparable operation occurring after $o$ in $l'$ is $rm(z_2)$ (they are surrounded by boxes in the figure). In this process, whether a $p$-incomparable operation is in $W'$ or not only relies on whether it is before or after such an $o$ in $l'$.

2. The second step is to define $U'$ and $V'$. $U'$ contains two kinds of operations: (1) operations whose linearization points are before $\text{ret}(put, x, p)$, and (2) other $put$ operations with priority $p$. $V'$ contains the remaining operations. In this example, $U'$ contains $put(z_1, p_2)$ and $put(x_2, p)$.

In conclusion, we have that $l_i'' = put(z_1, p_2) \cdot put(x_2, p)$, $l_j'' = put(z_2, p_2) \cdot rm(x_2) \cdot put(y_1, p_1) \cdot rm(y_1)$, and $l_3'' = rm(z_1) \cdot rm(z_2)$. Figure 1(b) draws linearization points for each operation in $e$ defining the linearization $l''$.

Section 4 introduces a characterization of concurrent priority queue violations using a set of non-recursive automata (whose states consist of a fixed number of registers), whose standard synchronized product is equivalent to Check-PQ-Conc (modulo a renaming of values which is possible by data-independence). Since SeqPQ is closed under projection (Lemma 3), the recursion in Check-PQ-Conc can be eliminated by checking that each projection of a given execution $e$ passes a non-recursive version of Check-PQ-Conc where every recursive call return Check-PQ-Conc(...) is replaced by return true. Let Check-PQ-Conc-NonRec be the thus obtained procedure.

| Lemma 7. | Given a data-differentiated execution $e$, $e \sqsubseteq \text{SeqPQ}$ if and only if for each $e' \in \text{proj}(e)$, Check-PQ-Conc-NonRec($e'$) returns true. |

4 Reducing Linearizability of Priority Queues to Reachability

We show that the set of executions for which Check-PQ-Conc-NonRec fails on some projection can be described using register automata, modulo a value renaming. Renaming values (which is complete under data independence) allows to simplify the reasoning about projections. W.l.o.g., we assume that all the operations which are not in the projection failing this test use the same distinguished value $\top$, different from those in the projection. Then, it is enough to find an automata characterization of the executions $e$ for which
Given a data-differentiated execution, the set of values with the same priority are observed. More precisely, we assume that every implementation results in an automaton satisfying the following property called $\Gamma$-completeness.

**Definition 8.** For $\Gamma \in \{\text{EmptyRemove, UnmatchedMaxPriority, MatchedMaxPriority}\}$, an automaton $A$ is called $\Gamma$-complete when for each data-independent implementation $I$: $A \cap I \neq \emptyset$ iff there exists $e \in I$ and $e' \in \text{proj}(e)$ such that $e'$ is not $\Gamma$-linearizable.

We can show that for any $\Gamma \in \{\text{EmptyRemove, UnmatchedMaxPriority, MatchedMaxPriority}\}$ there exists a $\Gamma$-complete automaton. For lack of space, we only consider the case $\Gamma = \text{MatchedMaxPriority}$ in Section 4.1. When defining $\Gamma$-complete automata, we assume that every implementation $I$ behaves correctly, i.e., as a FIFO queue, when only values with the same priority are observed. More precisely, we assume that for every execution $e \in I$ and every priority $p \in P$, the projection of $e$ to values with priority $p$ is linearizable (w.r.t. SeqPQ). This property can be checked separately using register automata similar to the automata in [5] describing FIFO queue violations. This assumption excludes some obvious violations, such as an $\text{rm}(a)$ operation happening before a $\text{pu}(a, p)$ operation, for some $p$.

For $\Gamma \in \{\text{UnmatchedMaxPriority, MatchedMaxPriority}\}$, we consider $\Gamma$-complete automata recognizing executions which contain only one maximal priority. This is w.l.o.g. because any data-differentiated execution for which $\Gamma(e)$ is false has such a projection. Formally, given a data-differentiated execution $e$ and $p$ a maximal priority in $e$, $e|_{\preceq_p}$ is the projection of $e$ to the set of values with priorities smaller or equal to $p$. Then,

**Lemma 9.** For $\Gamma \in \{\text{UnmatchedMaxPriority, MatchedMaxPriority}\}$, a data-differentiated execution $e$ is $\Gamma$-linearizable iff $e|_{\preceq_p}$ is $\Gamma$-linearizable for some maximal priority $p$ in $e$.

**Proof.** (Sketch) For the “only-if” direction, let $e$ be a data-differentiated execution linearizable w.r.t. $l = u \cdot \text{pu}(x, p) \cdot v \cdot \text{rm}(x) \cdot w$ s.t. $\text{MatchedMaxPriority-Seq}(l, x)$ holds. Since the predicate $\text{MatchedMaxPriority-Seq}(l, x)$ imposes no restriction on the operations in $u$, $v$, and $w$ with priorities incomparable to $p$, erasing all these operations results in a sequential execution which still satisfies this predicate. Similarly, for $\Gamma = \text{UnmatchedMaxPriority}$.

The “if” direction follows from the fact that if the projection of an execution to a set of operations $O_1$ has a linearization $l_1$ and the projection of the same execution to the remaining set of operations has a linearization $l_2$, then the execution has a linearization which is defined as an interleaving of $l_1$ and $l_2$.

Thus, let $e$ be an execution such that $e|_{\preceq_p}$ is linearizable w.r.t. $l = u \cdot \text{pu}(x, p) \cdot v \cdot \text{rm}(x) \cdot w$ where $\text{MatchedMaxPriority-Seq}(l, x)$ holds. By the property above, we know that $e$ has a linearization $l' = u' \cdot \text{pu}(x, p) \cdot v' \cdot \text{rm}(x) \cdot w'$, such that the projection of $l'$ to values of priority incomparable to $p$ is $l$. Since $\text{MatchedMaxPriority-Seq}(l, x)$ doesn’t constrain the values of priority incomparable to $p$, we obtain that $\text{MatchedMaxPriority-Seq}(l', x)$ also holds.

The existence of $\Gamma$-complete automata enable an effective reduction of checking linearizability of concurrent priority queue implementations to state reachability. Section 4.2 discusses decidability results implied by this reduction.

**Theorem 10.** Let $I$ be a data-independent implementation. Then, there is a $\Gamma$-complete automaton $A(\Gamma)$ for each $\Gamma$. Moreover, $I \subseteq \text{SeqPQ}$ iff $I \cap A(\Gamma) = \emptyset$ for all $\Gamma$. 

CONCUR 2017
Checking Linearizability of Concurrent Priority Queues

\[ \text{put}(b, p_4) \quad \text{rm}(b) \]
\[ \text{put}(c, p_1) \quad \text{rm}(c) \]
\[ \text{put}(d, p_2) \quad \text{rm}(d) \]
\[ \text{put}(f, p_3) \quad \text{rm}(f) \]

Figure 2 An execution that is not MatchedMaxPriority\textsuperscript{2}-linearizable. We represent each operation as a time interval whose left, resp., right, bound corresponds to the call, resp., return action.

4.1 A MatchedMaxPriority-complete automaton

A differentiated execution \( e \) is not MatchedMaxPriority-linearizable when all the put operations in \( e \) using the maximal priority \( p \) are matched, and \( e \) is not linearizable w.r.t. the set of sequential executions satisfying MatchedMaxPriority-Seq\((e, x)\) for each value \( x \) of priority \( p \). We consider two cases depending on whether \( e \) contains exactly one value with priority \( p \) or at least two values. We denote by MatchedMaxPriority\textsuperscript{>\(p\)} the strengthening of MatchedMaxPriority with the condition that all the values other than \( x \) have a priority strictly smaller than \( p \) (corresponding to the first case), and by MatchedMaxPriority\textsuperscript{=\(p\)} the strengthening of the same formula with the negation of this condition (corresponding to the second case). We use particular instances of register automata [15, 9, 18] whose states include only two registers, one for storing a priority guessed at the initial state, and one for storing the priority of the current action in the execution. The transitions can check equality or the order relation \( \prec \) between the values stored in the two registers. Instead of formalizing the full class of register automata, we consider a simpler class which suffices our needs. Thus, we consider a class of labeled transition systems whose states consist of a finite control part and a register \( r \) interpreted to elements of \( \mathbb{P} \). The transition labels are:

- \( r = \ast \) for storing an arbitrary value to \( r \),
- \( \text{call}(\text{rm}, a) \) and \( \text{ret}(\text{rm}, a) \) for reading call/return actions of a remove,
- \( \text{call}(\text{put}, d, g) \) where \( g \in \{= r, \prec r, \text{true}\} \) is a guard, for reading a call action \( \text{call}(\text{put}, d, p) \) and checking if \( p \) is either equal to or smaller than the value stored in \( r \), or arbitrary,
- \( \text{ret}(\text{put}, d, \text{true}) \) for reading a return action \( \text{ret}(\text{put}, d, p) \) for any \( p \).

The set of sequences (executions) accepted by such a transition system is defined as usual.

4.1.1 A MatchedMaxPriority\textsuperscript{>\(p\)}-complete automaton

Figure 2 contains a typical example of an execution \( e \) which is not MatchedMaxPriority\textsuperscript{>\(p\)}-linearizable, where \( p_1 \prec p_1, p_2 \prec p_4, \) and \( p_3 \prec p_4 \). Intuitively, this is a violation because during the whole execution of \( \text{rm}(b) \), the priority queue stores a smaller priority value (which should be removed before \( b \)). To be more precise, we define the interval of a value \( x \) as the time interval from the return of a put \( \text{ret}(\text{put}, x, p) \) to the call of the matching remove \( \text{call}(\text{rm}, x) \), or to the end of the execution if such a call action doesn’t exist. This represents the time interval in which a value is guaranteed to be stored into the priority queue. Concretely, for a standard indexing of actions in an execution, a time interval is a closed interval between the indexes of two actions in the execution. In Figure 2, the interval of each value of priority smaller than \( p_4 \) is pictured as a dashed line. There is no sequence \( l \) s.t. \( e \subseteq l \) and MatchedMaxPriority-Seq\((l, b)\) hold, since each time point from \( \text{call}(\text{rm}, b) \) to \( \text{ret}(\text{rm}, b) \) is included in the interval of a smaller priority value, and \( \text{rm}(b) \) can’t take effect in the interval of a smaller priority value. To formalize this scenario we use the notion of left-right constraint defined below.
Definition 11. Let $e$ be a data-differentiated execution which contains only one maximal priority $p$, and only one value $x$ of priority $p$ (and no $\text{rm}(\text{empty})$ operations). The left-right constraint of $e$ is the graph $G$ where:

- the nodes are the values occurring in $e$,
- there is an edge from $d_1$ to $x$, if $\text{put}(d_1, \_)<_{hb} \text{put}(x, p)$ or $\text{put}(d_1, \_)<_{hb} \text{rm}(x)$,
- there is an edge from $x$ to $d_1$, if $\text{rm}(x)<_{hb} \text{rm}(d_1)$ or $\text{rm}(d_1)$ does not exists,
- there is an edge from $d_1$ to $d_2$, if $\text{put}(d_1, \_)<_{hb} \text{rm}(d_2, \_)$.

The execution in Figure 2 is not MatchedMaxPriority-complete because the left-right constraint of the maximal priority value $b$ contains the cycle $f \rightarrow d \rightarrow c \rightarrow b \rightarrow f$. The presence of such a cycle is equivalent to non MatchedMaxPriority-complete linearity.

Lemma 12. Let $e$ be a data-differentiated execution such that $\text{Has-MatchedMaxPriority}(c)$ holds, $p$ is the maximal priority in $e$, and $\text{put}(x, p)$ and $\text{rm}(x)$ are only operations with arguments of priority $p$ in $e$. Then, $e$ is MatchedMaxPriority-linearizable iff the left-right constraint of $x$ contains no cycle going through $x$.

When the left-right constraint contains a cycle $d_1 \rightarrow \ldots \rightarrow d_m \rightarrow x \rightarrow d_1$, for some $d_1, \ldots, d_m \in \mathbb{D}$, we say that $x$ is covered by $d_1, \ldots, d_m$. The shape of an execution witnessing such a cycle (i.e., the alternation between call/return actions) can be identified using our class of automata, the only complication being the unbounded number of values $d_1, \ldots, d_m$. However, by data independence, whenever an implementation contains such an execution it also contains an execution where all the values $d_1, \ldots, d_m$ are renamed to the same value $a$, and $x$ is renamed to $b$. Therefore, our automata can be defined over a fixed set of values $a$, $b$, and $\top$ (recall that $\top$ is used for operations outside of the non-linearizable projection).

To define a MatchedMaxPriority-complete automaton, we need to consider all the possible orders between the call/return actions of the put/rm operations that add and respectively, remove the value $b$. The case where the put happens-before the remove (as in Figure 2) is pictured in Figure 4. This automaton captures the three possible ways of ordering the first action $\text{ret}(\text{put}, a, \_)$ w.r.t. the actions with value $b$, which are pictured in Figure 3(a) (this action cannot occur after $\text{call}(\text{rm}, b, \_)$ since $b$ must be covered by the $a$-s). The paths corresponding to these three possible orders are: $q_1 \rightarrow q_2 \rightarrow q_3 \rightarrow \ldots \rightarrow q_7$, $q_1 \rightarrow q_2 \rightarrow q_3 \rightarrow \ldots \rightarrow q_{10}$, and $q_1 \rightarrow q_9 \rightarrow q_{10} \rightarrow \ldots \rightarrow q_7$. Figure 3 lists the four possible orderings of the call/return actions of adding and removing $b$, and also possible orders of the first $\text{ret}(\text{put}, a, \_)$ w.r.t. the actions with value $b$. Each such ordering corresponds to an automaton similar to the one in Figure 4, their union defining a MatchedMaxPriority-complete automaton.

4.1.2 A MatchedMaxPriority-complete automaton

When an execution contains at least two values of maximal priority, the acyclicity of the left-right constraints (for all the maximal priority values) is not enough to conclude that
The execution is MatchedMaxPriority-linearizable. Intuitively, there may exist a value \( a \) which is added before another value \( b \) such that all the possible linearization points of \( rm(b) \) are disabled by the position of \( rm(a) \) in the happens-before. We give an example of such an execution \( e \) in Figure 5, where \( p_1 < p_4 \). This execution is not linearizable w.r.t. MatchedMaxPriority (or MatchedMaxPriority\({}^\top\)) even if neither \( a \) nor \( b \) are covered by values with smaller priority. Since \( \text{put}(a, p_4) <_{hb} \text{put}(b, p_4) \) and values of the same priority are removed in FIFO order, \( \text{rm}(a) \) should be linearized before \( \text{rm}(b) \) (i.e., this execution should be linearizable w.r.t. a sequence where \( \text{rm}(a) \) occurs before \( \text{rm}(b) \)). Since \( \text{rm}(b) \) cannot take effect during the interval of a smaller priority value, it could be only linearized in one of the two time intervals pictured with dotted lines in Figure 5. However, each of those time intervals ends before \( \text{call}(\text{rm}, a) \), and thus \( \text{rm}(a) \) cannot be linearized before \( \text{rm}(b) \).

To recognize the scenarios in Figure 5, we introduce an order \( <_{pb} \) between values which intuitively, can be thought of as “a value \( a \) is put before another value \( b \)”. More precisely, given a data-differentiated execution \( e \) and two values \( a \) and \( b \) of maximal priority, \( a <_{pb} b \) if one of the following holds: (1) \( \text{put}(a, \_\_\_) <_{hb} \text{put}(b, \_\_\_) \), (2) \( \text{rm}(a) <_{hb} \text{rm}(b) \), or (3) \( \text{rm}(a) <_{hb} \text{rm}(b, \_\_\_) \). Sometimes we use \( a <_{pb}^A b \), \( a <_{pb}^B b \), and \( a <_{pb}^C b \) to explicitly distinguish between these three cases. Let \( <_{pb}^* \) be the transitive closure of \( <_{pb} \).

To define the time intervals in which a remove like \( \text{rm}(b) \) in Figure 5, can be linearized (outside of intervals of smaller priority values) we use the notion of gap-point. As before, defining time intervals relies on an indexing of actions in an execution, starting with 0.

**Definition 13.** Let \( e \) be a data-differentiated execution with only one maximal priority \( p \), and \( \text{put}(x, p) \) and \( \text{rm}(x) \) two operations in \( e \). An index \( i \in [0, |e| - 1] \) is a gap-point of \( x \) if \( i \) is greater than or equal to the index of both \( \text{call}(\text{put}, x, p) \) and \( \text{call}(\text{rm}, x) \), smaller than the index of \( \text{ret}(\text{rm}, x) \), and not included in the interval of a value with priority smaller than \( p \).

The case of Figure 5 can be formally described as follows: \( a <_{pb}^A b \) while the right-most gap-point of \( b \) is before \( \text{call}(\text{rm}, a) \) or \( \text{call}(\text{put}, a, p_4) \). The following lemma states that these conditions are enough to characterize non-linearizability w.r.t. MatchedMaxPriority\({}^\top\).
We omit the operations which can be ordered arbitrarily, e.g., put(b) in the cases (d) and (e).

**Figure 6** Orderings to be considered when defining a MatchedMaxPriority-pr-complete automaton. We omit the operations which can be ordered arbitrarily, e.g., put(b) in the cases (d) and (e).

**Figure 7** A register automaton for the case in Figure 6(a), where \( A_1 = A \cup \{ \text{call}(\text{put}, c, r) \}, A_2 = A_1 \cup \{ \text{ret}(\text{put}, b, = r) \}, A_3 = A_2 \cup \{ \text{ret}(\text{rm}, b) \}, A_4 = A_3 \cup \{ \text{ret}(\text{put}, b, = r) \}, \text{ret}(\text{rm}, c) \}, \) where \( A = \{ \text{call}(\text{put}, \top, \text{true}), \text{ret}(\text{put}, \top, \text{true}), \text{call}(\text{rm}, \top), \text{ret}(\text{rm}, \top), \text{call}(\text{rm}, \text{empty}), \text{ret}(\text{rm}, \text{empty}) \}. \)

**Lemma 14.** Let \( e \) be a data-differentiated execution with only one maximal priority \( p \) such that Has-MatchedMaxPriority\(^p\) holds. Then, \( e \) is not MatchedMaxPriority\(^p\)-linearizable iff \( e \) contains two values \( x \) and \( y \) of maximal priority \( p \) such that \( y <^p x \), and the rightmost gap-point of \( x \) is strictly smaller than the index of call(put, y, p) or call(rm, y).

The following shows that the number of values needed to witness that \( y <^p x \), for some \( x \) and \( y \), is bounded.

**Lemma 15.** Let \( e \) be a data-differentiated execution such that \( a <^p a_1 <^p \ldots <^p a_m <^p b \) holds for some set of values \( a, a_1, \ldots, a_m, b \). Then, one of the following holds:

- \( a <^p b, a <^p b, \) or \( a <^p b, \)
- \( a <^p a_i <^p b \) or \( a <^p a_i <^p b, \) for some \( i \).

To characterize violations to MatchedMaxPriority\(^p\)-linearizability, one has to consider all the possible orders between call/return actions of the operations on values \( a, b, \) and \( a_i \) in Lemma 15, and the right-most gap point of \( b \). Excluding the inconsistent cases, we are left with the five orders in Figure 6, where \( o \) denotes the rightmost gap-point of \( b \). For each case, we define an automaton recognizing the induced set of violations. The register automaton for the case in Figure 6(a) is shown in Figure 7. In this case, Lemma 14 is equivalent to the fact that intuitively, the time interval from call(rm, a) to ret(rm, b) is covered by lower priority values (thus, there is no gap-point of \( b \) which occurs after call(rm, a)). By data-independence, these lower priority values can be renamed to a fixed value \( c \).
4.2 Decidability Result

We describe a class $C$ of data-independent implementations for which linearizability w.r.t. $\text{SeqPQ}$ is decidable. The implementations in $C$ allow an unbounded number of values but a bounded number of priorities. Each method has a finite set of local variables storing Boolean values or values from $\mathbb{D}$. Methods communicate through a finite number of shared variables interpreted also as Booleans or values from $\mathbb{D}$. To ensure data independence, values in $\mathbb{D}$ may be assigned, but never used in Boolean expressions (e.g., of if-then-else statements). This class captures typical implementations, or finite-state abstractions thereof, e.g., obtained via predicate abstraction. The $\Gamma$-complete automata we define use a fixed set $D = \{a, b, c, a_1, \top\}$ of values ($a_1$ is needed to deal with the second item in Lemma 15). Therefore, for any $\Gamma$, $C \cap A(\Gamma) \neq \emptyset$ iff $C_D \cap A(\Gamma) \neq \emptyset$, where $C_D$ is the subset of $C$ that uses only values in $D$.

The set of executions $C_D$ can be represented by a Vector Addition System with States (VASS). Since the set of values and priorities is bounded, each method invocation can be represented by a finite-state automaton (see [4]). For a fixed set of priorities $P \subseteq \mathbb{P}$, the register automata $A(\Gamma)$ can be transformed to finite-state automata (the number of valuations of the registers is bounded). Thus, checking linearizability of an implementation in $C$ is PSPACE when the number of threads is bounded, and EXPSpace, otherwise. Moreover, reachability in VASSs can be reduced to checking linearizability of such an implementation. Essentially, given an instance of the VASS reachability problem, one can define a priority queue implementation where the put methods behave correctly and additionally, they include the code of the VASS simulation defined in [4], and the rm methods behave correctly, except for the moment where the target state is reached, in which case they trigger a linearizability violation by returning an arbitrary value.

$\triangleright$ Theorem 16. Verifying whether an implementation in $C$ is linearizable w.r.t. $\text{SeqPQ}$ is PSPACE-complete for a fixed number of threads, and EXPSpace-complete otherwise.

5 Related work

The theoretical limits of checking linearizability have been investigated in previous works. Checking linearizability of a single execution w.r.t. an arbitrary ADT is NP-complete [11] while checking linearizability of all the executions of a finite-state implementation w.r.t. an arbitrary ADT specification (given as a regular language) is EXPSpace-complete when the number of program threads is bounded [3, 12], and undecidable otherwise [4].

Existing automated methods for proving linearizability of a concurrent object are also based on reductions to safety verification, e.g., [1, 13, 21]. The approach in [21] considers implementations where operations’ linearization points are manually specified. Essentially, this approach instruments the implementation with ghost variables simulating the ADT specification at linearization points. This approach is incomplete since not all implementations have fixed linearization points. Aspect-oriented proofs [13] reduce linearizability to the verification of four simpler safety properties. This approach has only been applied to queues, and has not produced a fully automated and complete proof technique. The work in [10] proves linearizability of stack implementations with an automated proof assistant. Their approach does not lead to full automation however, e.g., by reduction to safety verification.

Our previous work [5] shows that checking linearizability of finite-state implementations of concurrent queues and stacks is decidable. Roughly, we follow the same schema: the recursive procedure in Section 3.1 is similar to the inductive rules in [5], and its extension to concurrent executions in Section 3.2 corresponds to the notion of step-by-step linearizability in [5]. Although similar in nature, defining these procedures and establishing their
correctness require proof techniques which are specific to the priority queue semantics. The order in which values are removed from a priority queue is encoded in their priorities which come from an unbounded domain, and not in the happens-before order as in the case of stacks and queues. Therefore, the results we introduce in this paper cannot be inferred from those in [5]. At a technical level, characterizing the priority queue violations requires a more expressive class of automata (with registers) than the finite-state automata in [5].

References


Checking Linearizability of Concurrent Priority Queues


Nash Equilibrium and Bisimulation Invariance

Julian Gutierrez¹, Paul Harrenstein², Giuseppe Perelli³, and Michael Wooldridge⁴

¹ Department of Computer Science, University of Oxford, Oxford, United Kingdom
julian.gutierrez@cs.ox.ac.uk
² Department of Computer Science, University of Oxford, Oxford, United Kingdom
paul.harrenstein@cs.ox.ac.uk
³ Department of Computer Science, University of Oxford, Oxford, United Kingdom
giuseppe.perelli@cs.ox.ac.uk
⁴ Department of Computer Science, University of Oxford, Oxford, United Kingdom
mjw@cs.ox.ac.uk

Abstract

Game theory provides a well-established framework for the analysis of concurrent and multi-agent systems. The basic idea is that concurrent processes (agents) can be understood as corresponding to players in a game; plays represent the possible computation runs of the system; and strategies define the behaviour of agents. Typically, strategies are modelled as functions from sequences of system states to player actions. Analysing a system in such a way involves computing the set of (Nash) equilibria in the game. However, we show that, with respect to the above model of strategies—the standard model in the literature—bisimilarity does not preserve the existence of Nash equilibria. Thus, two concurrent games which are behaviourally equivalent from a semantic perspective, and which from a logical perspective satisfy the same temporal formulae, nevertheless have fundamentally different properties from a game theoretic perspective. In this paper we explore the issues raised by this discovery, and investigate three models of strategies with respect to which the existence of Nash equilibria is preserved under bisimilarity. We also use some of these models of strategies to provide new semantic foundations for logics for strategic reasoning, and investigate restricted scenarios where bisimilarity can be shown to preserve the existence of Nash equilibria with respect to the conventional model of strategies in the literature.

1998 ACM Subject Classification F.3.1 Specifying and Verifying and Reasoning about Programs, F.4.1 Mathematical logic (Temporal Logic), I.2.11 Distributed AI (Multiagent Systems)

Keywords and phrases Bisimulation, Nash Equilibrium, Multiagent Systems, Strategy Logic

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2017.17

1 Introduction

The concept of bisimilarity plays a central role in both the theory of concurrency [18, 16] and logic [26, 16]. In the context of concurrency, bisimilar systems are regarded as behaviourally equivalent—appearing to have the same behaviour when interacting with an

* The authors acknowledge with gratitude the financial support of the ERC Advanced Investigator grant 291528 (“RACE”) at Oxford.
arbitrary environment. From a logical/verification perspective, bisimilar systems are known to satisfy the same temporal logic properties with respect to languages such as LTL, CTL, or the $\mu$-calculus. These features, in turn, make it possible to verify temporal logic properties of concurrent systems using bisimulation-based approaches. For example, temporal logic model checking techniques may be optimised by applying them to the smallest bisimulation-equivalent model of the system being analysed; or, indeed, to every other model within the system’s bisimulation equivalence class. This is possible because the properties that one is interested in checking are bisimulation invariant.

Model checking is not the only verification technique that can benefit from bisimulation invariance. Consider abstraction and refinement techniques [9, 10] (where a set of states is either collapsed or broken down in order to build a somewhat simpler set of states); coinduction methods [25] (which can be used to check the correctness of an implementation with respect to a given specification); or reduced BDD representations of a system [7] (where isomorphic, and therefore bisimilar, subgraphs are merged, thereby eliminating part of the initial state space of the system). Bisimulation invariance is therefore a very important concept in the formal analysis and verification of concurrent and multi-agent systems.

Game theory [22] provides another important framework for the analysis of concurrent and multi-agent systems. Within this framework, a concurrent/multi-agent system is viewed as a game, where processes/agents correspond to players, system executions/computation runs to plays, and individual process behaviours are modelled as player strategies, which are used to resolve the possible nondeterministic choices available to each player. In logic and computer science, games have also been extensively used for synthesis and verification. In this case, one is usually focused on two-player zero-sum games where the desired solution concept is given by the existence of winning strategies. Instead, in this paper, we are more interested in the more general framework given by multi-player non-zero-sum games, where the standard solution concept is given by of strategies forming a Nash equilibrium.

A widely-used model for strategies in (concurrent) multi-player games is to view a strategy for a process $i$ as a function $f_i$ which maps finite histories $s_0, s_1, \ldots, s_k$ of system states to actions $f_i(s_0, s_1, \ldots, s_k)$ available to the process/agent/player $i$ at state $s_k$. In what follows, we use the terms process, agent, and player interchangeably. We refer to this as the conventional model of strategies, as it is the best-known and most widely used model in logic, AI, and computer science (and indeed in extensive form games [22]). For instance, specification languages such as alternating-time temporal logic (ATL [4]), and formal models such as concurrent game structures [4] use this model of strategies. If we model a concurrent/multi-agent system as a game in this way, then the analysis and verification of the system reduces to computing the set of (Nash) equilibria in the associated multi-player game; in some cases, the analysis reduces to the computation of a winning strategy.

Because bisimilar systems are regarded as behaviourally equivalent, and bisimilar systems satisfy the same set of temporal logic properties, it is natural to ask whether the Nash equilibria of bisimilar structures are identical as well; that is, we ask the following question:

Is Nash equilibrium invariant under bisimilarity?

We show that, for the conventional model of strategies, the answer to this question is, in general, ‘no’. More specifically, the answer critically depends on precisely how players’ strategies are modelled. With the conventional model of strategies, the answer is positive only for some two-player games, but negative in general for games with more than two players. This means, for instance, that, in the general case, bisimulation-based techniques cannot be used when one is also reasoning about the Nash equilibria of concurrent systems that are formally modelled as (concurrent) multi-player games.
Motivated by this observation—which brings together in a striking way a fundamental concept in game theory and a fundamental concept in logic/concurrency—the purpose of the present paper is to investigate this issue in detail. We present three alternative models of strategies in which Nash equilibria and their existence are preserved under bisimilarity. We also study the above question for special classes of systems, and explore applications to logic. Specifically, we investigate the implications of replacing the conventional model of strategies with some of the models we propose in this paper in logics for strategic reasoning [19, 8], in particular, the semantic implications with respect to Strategy Logic (SL [19]). We also show that, within the conventional model of strategies, Nash equilibrium is preserved by bisimilarity in certain two-player games as well as in the class of concurrent game structures that are induced by iterated Boolean games [14], a logic-based framework for reasoning about the strategic behaviour of different kinds of multi-agent systems [28, 29].

A Motivating Example

So far we have mentioned some cases where one needs or desires a property to be invariant under bisimilarity. However, one may still wonder why it is so important that the particular property of having a Nash equilibrium is preserved under bisimilarity. One reason has its roots in automated formal verification. To illustrate this, imagine that the system of Figure 1 is given as input to a verification tool. It is likely that such a tool will try to perform as many optimisations as possible to the system before any analysis is performed. The simplest of such optimisations—for instance as done by virtually every model checking tool—is to reduce the input system by merging isomorphic subtrees (e.g., when generating the ROBDD representation of a system). If such an optimisation is made, the tool will construct the (bisimilar) system in Figure 2. (Observe that the subgraphs rooted at $s_1$ and $s'_1$ are isomorphic.) However, with respect to the existence of Nash equilibria, such a transformation is unsound in the general case.

For instance, suppose that the system in Figure 1 represents a 3-player game, where each transition is labelled by the choices $x, y, z$ made by player 1, 2, and 3, respectively, an asterisk $*$ being a wildcard for any action for the player in the respective position. Thus, whereas players 1 and 2 can choose to play either $a$ or $b$ at each state, player 3 can choose between $a, b, a'$, or $b'$. The states are labelled by valuations $xy$ over $\{p, q\}$, where $\bar{x}$ indicates that $x$ is set to false. Assume that player 1 would like $p$ to be true sometime, that player 2...
would like \( q \) to be true sometime, and that player 3 desires to prevent both player 1 and player 2 from achieving their goals. Accordingly, their preferences/goals can be formally represented by the LTL formulae  
\[
\gamma_1 = Fp, \quad \gamma_2 = Fq, \quad \gamma_3 = G\neg(p \lor q),
\]
respectively. Informally, \( F\varphi \) means “eventually \( \varphi \) holds” and \( G\varphi \) means “always \( \varphi \) holds.” Moreover, given these players’ goals and the conventional model of strategies, we will see later in Section 4.2 that the system in Figure 1 has a Nash equilibrium, whereas no Nash equilibria exists in the (bisimilar) system in Figure 2. This example illustrates a major issue when analysing Nash equilibria in the most widely used models of strategies and games studied in the literature, namely, that even the simplest optimisations commonly used in automated formal verification are not sound with respect to game theoretic analyses.

2 Preliminaries

We begin by introducing the main technical concepts and models used in this paper.

Concurrent Game Structures

We use the model of concurrent game structures, which are well-established in the literature (see, for instance, [4]). A concurrent game structure (CGS) is a tuple \( M = (Ag, AP, Ac, St, s_0^M, \lambda, \delta) \), where \( Ag = \{1, \ldots, n\} \) is a set of players or agents, \( AP \) a set of propositional variables, \( Ac \) is a set of actions, \( St \) is a set of states containing a unique initial state \( s_0^M \). With each player \( i \in Ag \) and each state \( s \in St \), we associate a non-empty set \( Ac_i(s) \) of feasible actions that, intuitively, \( i \) can perform when in state \( s \). By a direction or decision we understand a profile of actions \( d = (a_1, \ldots, a_n) \) in \( Ac \times \cdots \times Ac \) and we let \( \text{Dir} \) denote the set of directions. A direction \( d = (a_1, \ldots, a_n) \) is legal at state \( s \) if \( a_i \in Ac_i(s) \) for all players \( i \). Unless stated otherwise, by “direction” we will henceforth generally mean “legal direction”. Furthermore, \( \lambda: St \to 2^{\mathcal{AP}} \) is a labelling function, associating with every state \( s \) a valuation \( v \in 2^\mathcal{AP} \). Finally, \( \delta \) is a deterministic transition function, which associates with each state \( s \) and every legal direction \( d = (a_1, \ldots, a_n) \) at \( s \) a state \( \delta(a_1, \ldots, a_n) \). As such \( \delta \) characterises the behaviour of the system when \( d = (a_1, \ldots, a_n) \) is performed at state \( s \).

Computations, Runs, and Traces

The possible behaviours exhibited by a concurrent game structure can be described at at least three different levels of abstraction. Thus, we distinguish between computations, runs, and traces. Computations carry the most information, while traces carry the least, in the

**Figure 2** A 3-player game without Nash equilibria.
sense that every computation induces a unique run and every run induces a unique trace, but not necessarily the other way round.

A state \( s' \) is accessible from another state \( s \) whenever there is some \( d = (a_1, \ldots, a_n) \) such that \( d \) is legal at \( s \) and \( \delta(s, a_1, \ldots, a_n) = s' \). For easy readability we then also write \( s \overset{d}{\longrightarrow} s' \).

An (infinite) computation is then an infinite sequence of directions \( \kappa = d_0, d_1, d_2, \ldots \) such that there are states \( s_0, s_1, \ldots \) such that \( s_0 = s_0^M \) and \( s_0 \overset{d_0}{\longrightarrow} s_1 \overset{d_1}{\longrightarrow} s_2 \overset{d_2}{\longrightarrow} \cdots \). Observe that in every concurrent game model the states \( s_0, s_1, \ldots \) in the above definition always exist and are unique. A finite computation is finite prefix of a computation \( \kappa \). We also allow a finite computation to be the empty sequence \( \epsilon \) of directions. We also use \( \delta^*(s, d_0, d_1, \ldots, d_k) \) to denote the unique state that is reached from the state \( s \) after applying the computation \( d_0, d_1, \ldots, d_k \).

An (infinite) run is an infinite sequence \( \rho = s_0, s_1, s_2, \ldots \) of states of sequentially accessible states, with \( s_0 = s_0^M \). We say that run \( s_0, \ldots, s_k \) is induced by computation \( d_0, \ldots, d_{k-1} \) if \( s_0 \overset{d_0}{\longrightarrow} s_1 \overset{d_1}{\longrightarrow} s_2 \overset{d_2}{\longrightarrow} \cdots \) and \( s_0 = s_0^M \). Thus, every computation induces a unique run and every run is induced by at least one computation. A finite run is a finite prefix of a run. The sets of infinite and finite runs are denoted by \( \text{runs}_M^\omega \) and \( \text{runs}_M \), respectively.

An (infinite) trace is a sequence \( \tau = v_0, v_1, v_2, \ldots \) of valuations such that there is a run \( \rho = s_0, s_1, s_2, \ldots \) in \( \text{runs}_M^\omega \) such that \( v_k = \lambda(s_k) \) for every \( k \geq 0 \), that is, \( \tau = \lambda(s_0), \lambda(s_1), \lambda(s_2), \ldots \). In that case we also say that trace \( \tau \) is induced by run \( \rho \), and if \( \rho \) is induced by computation \( \kappa \), also that \( \tau \) is induced by \( \kappa \). By a finite trace we mean a finite prefix of a trace. We denote the sets of finite and infinite traces of a concurrent game structure \( M \) by \( \text{traces}_M \) and \( \text{traces}_M^\omega \), respectively. We use \( \rho_M(\kappa) \) to denote the run induced by an infinite computation \( \kappa \) in \( M \), and \( \pi_M(\kappa) \), if \( \kappa \) is finite, on the understanding that \( \pi_M(\kappa) = s_0^M \). Also, if \( \rho = s_0, s_1, s_2, \ldots \) is a run, by \( \tau_M(\rho) \) we denote the trace \( \lambda(s_0), \lambda(s_1), \lambda(s_2), \ldots \), and similarly for finite runs \( \tau \in \text{runs}_M \). Finally, \( \tau_M(\rho_M(\kappa)) \) is abbreviated as \( \tau_M(\kappa) \). When no confusion is likely, we sometimes omit the subscript \( M \) and the qualification ‘finite’.

Bisimulations and Bisimilarity

One of the most important behavioural/observational equivalences in concurrency is bisimilarity, which is usually defined over Kripke structures or labelled transition systems (see, for instance, [18, 16]). However, the equivalence can be uniformly defined for general concurrent game structures, where decisions/directions play the role of “actions” in transition systems. Formally, let \( M = (\text{AP}, \text{Ag}, \text{Ac}, \text{St}, s_0^M, \lambda, \delta) \) and \( M' = (\text{AP}, \text{Ag}, \text{Ac}, \text{St}', s_0^M, \lambda', \delta') \) be two concurrent game structures. A bisimulation, denoted by \( \sim \), between states \( s^* \in \text{St} \) and \( t^* \in \text{St}' \) is a non-empty binary relation \( R \subseteq \text{St} \times \text{St}' \), such that \( s^* \text{ R } t^* \) and for all \( s, s' \in \text{St} \), \( t, t' \in \text{St}' \), and \( d \in \text{Dir} \):

\[
\begin{align*}
& s \text{ R } t \text{ implies } \lambda(s) = \lambda'(t), \\
& s \text{ R } t \text{ and } s \overset{d}{\longrightarrow} s' \text{ implies } t \overset{d}{\longrightarrow} t'' \text{ for some } t'' \in \text{St}' \text{ with } s' \text{ R } t'', \\
& s \text{ R } t \text{ and } t \overset{d}{\longrightarrow} t' \text{ implies } s \overset{d}{\longrightarrow} s'' \text{ for some } s'' \in \text{St} \text{ with } s' \text{ R } t'.
\end{align*}
\]

Then, if there is a bisimulation between two states \( s^* \) and \( t^* \), we say that they are bisimilar and write \( s^* \sim t^* \) in such a case. We also say that concurrent game structures \( M \) and \( M' \) are bisimilar—in symbols \( M \sim M' \)—if \( s_0^M \sim s_0^{M'} \). We furthermore say that runs \( \rho = s_0, s_1, \ldots \) and \( \rho' = s_0', s_1', \ldots \) are statewise bisimilar (in symbols \( \rho \sim \rho' \)) if \( s_k \sim s_k' \) for every \( k \geq 0 \). Both bisimilarity and statewise bisimilarity are equivalence relations, which is a standard result in the literature (see, for instance, [11, 5, 18]). We, moreover, find that the sets of (finite) computations as well as the sets of (finite) traces of two bisimilar concurrent game structures are identical. Moreover, every (finite) computation \( \kappa \) gives rise to statewise bisimilar (finite)
Nash Equilibrium and Bisimulation Invariance

runs and identical (finite) traces in bisimilar concurrent game structures. However, as runs are sequences of states and the states of different concurrent game structures \( M \) and \( M' \) may be distinct, even if they are bisimilar, no identification of their sets \( \text{runs}_n^M \) and \( \text{runs}_n^{M'} \) of runs can generally be made.

### 3 Games on Concurrent Game Structures

Concurrent game structures specify the actions the players can take at each state and which states are reached if they all concurrently decide on an action. A full understanding of the system that is being modelled, however, also depends on what goals the players desire to achieve and on what strategies they may adopt in pursuit of these goals. We therefore augment concurrent game structures with preferences and strategies for the players. Thus they define a strategic game and as such they are amenable to game theoretic analysis by standard solution concepts, among which Nash equilibrium is probably the most prominent.

#### 3.1 Strategies and Strategy Profiles

Based on the distinction between computations, runs, and traces, we can also distinguish three types of strategy: computation-based, run-based, and trace-based strategies. The importance of these distinctions is additionally corroborated by Bouyer et al. [6], who show how the specific model of strategies adopted affects the computational complexity of some standard decision problems related to multi-agent systems.

A computation-based strategy for a player \( i \) in a concurrent game structure \( M \) is a function \( f_i^{\text{comp}}: \text{comps}_M \to \text{Ac} \) such that \( f_i^{\text{comp}}(\kappa) \in \text{Ac}_i(s_k) \) for every finite \( \kappa \in \text{comps}_M \) with \( \pi_M(\kappa) = s_0, \ldots, s_k \). Thus, \( f_i^{\text{comp}}(\epsilon) \in \text{Ac}_i(s_0^\emptyset) \), where \( \epsilon \) is the empty sequence of directions.

Similarly, a run-based strategy for player \( i \) is a function \( f_i^{\text{run}}: \text{runs}_M \to \text{Ac} \) where \( f_i^{\text{run}}(s_0, \ldots, s_k) \in \text{Ac}_i(s_k) \) for every finite run \( (s_0, \ldots, s_k) \in \text{runs} \). Finally, a trace-based strategy for \( i \) is a function \( f_i^{\text{trace}}: \text{traces}_M \to \text{Ac} \) such that \( f_i^{\text{trace}}(\tau) \in \text{Ac}_i(s_k) \) for every trace \( \tau \in \text{traces}_M \) and every run \( \pi = s_0, \ldots, s_k \) such that \( \tau = \lambda(s_0), \ldots, \lambda(s_k) \).

A computation-based strategy profile is then a tuple \( f = (f_1, \ldots, f_n) \) that associates with each player \( i \) a computation-based strategy \( f_i \). Run-based and trace-based strategy profiles are defined analogously. Every computation-based strategy profile \( f = (f_1, \ldots, f_n) \) induces a unique computation \( \kappa_M(f) = d_0, d_1, d_2, \ldots \) in \( M \) that is defined inductively such that \( d_0 = (f_1(\epsilon), \ldots, f_n(\epsilon)) \) and \( d_{k+1} = (f_1(d_0, \ldots, d_k), \ldots, f_n(d_0, \ldots, d_k)) \), for all \( k \in \mathbb{N} \). Similarly, a run-based strategy profile \( f = (f_1, \ldots, f_n) \) defines the unique computation \( \kappa_M(f) = d_0, d_1, d_2, \ldots \) such that \( d_0 = (f_1(s_0^M), \ldots, f_n(s_0^M)) \) and \( d_{k+1} = (f_1(\pi(d_0, \ldots, d_k)), \ldots, f_n(\pi(d_0, \ldots, d_k))) \), for all \( k \in \mathbb{N} \). Finally, the computation \( \kappa_M(f) \) defined by a trace-based strategy profile \( f \) is such that \( d_0 = (f_1(\lambda(s_0^M)), \ldots, f_n(\lambda(s_0^M))) \) and \( d_{k+1} = (f_1(\tau(d_0, \ldots, d_k)), \ldots, f_n(\tau(d_0, \ldots, d_k))) \), for all \( k \in \mathbb{N} \). For \( f = (f_1, \ldots, f_n) \) a profile of computation-based, run-based, or trace-based strategies, we write with a slight abuse of notation \( \rho(f_1, \ldots, f_n) \) for \( \rho(\kappa(f_1, \ldots, f_n)) \) and \( \tau(f_1, \ldots, f_n) \) for \( \tau(\rho(f_1, \ldots, f_n)) \).

Note that, as the computations and traces of bisimilar concurrent game structures coincide, so do the computation-based and trace-based strategies available to the players. Moreover, the computations induced by them will be identical. With the states of bisimilar structures possibly being distinct, however, similar observations do not straightforwardly extend to run-based strategies and strategy profiles.
3.2 Preferences and Goals

We formally specify the preferences of a player $i$ of a concurrent game structure $M$ as a subset $\Gamma_i$ of computations, that is, $\Gamma_i \subseteq \text{comps}^\pi_M$, and refer to $\Gamma_i$ as player $i$’s goal set. Player $i$ is then understood to (strictly) prefer computations in $\Gamma_i$ to those not in $\Gamma_i$, and to be indifferent otherwise. Accordingly, each player’s preferences are dichotomous, only distinguishing between the preferred computations in $\Gamma_i$ and the not preferred ones not in $\Gamma_i$. Formally, player $i$ is said to weakly prefer computation $\kappa$ to computation $\kappa'$ if $\kappa \in \Gamma_i$ whenever $\kappa' \in \Gamma_i$, and to strictly prefer $\kappa$ to $\kappa'$ if $\kappa$ weakly prefers $\kappa$ to $\kappa'$ but not vice versa.

If player $i$ weakly prefers $\kappa$ to $\kappa'$ and $\kappa'$ to $\kappa$, we say that $i$ is indifferent between $\kappa$ and $\kappa'$.

The choice of modelling players’ preferences as sets of computations—rather than say sets of runs or sets of traces—is for technical convenience and flexibility. Observe that every set of runs is induced by a set of computations, namely the set of computations that give rise to the same runs, and similarly for every set of traces. Thus, we say that a goal set $\Gamma_i \subseteq \text{comps}^\pi_M$ is run-based if for any two computations $\kappa$ and $\kappa'$ with $\rho(\kappa) = \rho(\kappa')$ we have that $\kappa \in \Gamma_i$ if and only if $\kappa' \in \Gamma_i$. Similarly, $\Gamma_i$ is said to be trace-based whenever $\tau(\kappa) = \tau(\kappa')$ implies that $\kappa \in \Gamma_i$ if and only if $\kappa' \in \Gamma_i$. Thus, run-based goals are computation-based goals closed under induced runs, and trace-based goals are computation-based goals closed under induced traces.

Sometimes—as we did in the example in the introduction—players’ goals are specified by temporal logic formulae. As the satisfaction of goals only depends on traces, they will directly correspond to trace-based goals, given our formalisation of goals and preferences.

3.3 Games and Nash Equilibrium

With the above definitions in place, we now define a game on a concurrent game structure $M$ (also called a CGS-game) as a tuple, $G = (M, \Gamma_1, \ldots, \Gamma_n)$, where, for each player $i$, the goal set $\Gamma_i \subseteq \text{comps}^\pi_M$ specifies $i$’s dichotomous preferences over the computations in $M$.

The players of a CGS-game either all play computation-based strategies, all play run-based strategies, or all play trace-based strategies. For each such choice of type of strategies, with the set of players and their preferences specified, every CGS-game defines a strategic game in the standard game theoretic sense. Observe that the set of strategies is infinite in general. Thus the game theoretic solution concept of Nash equilibrium becomes available for the analysis of games on concurrent game structures. If $f = (f_1, \ldots, f_n)$ is a strategy profile and $g_i$ a strategy for player $i$, we write $(f_i, g_i)$ for the strategy profile $(f_1, \ldots, g_i, \ldots, f_n)$, which is identical to $f$ except that $i$’s strategy is replaced by $g_i$. Formally, given a CGS-game, we say that a profile $f = (f_1, \ldots, f_n)$ of computation-based strategies is a Nash equilibrium in computation-based strategies (or computation-based equilibrium) if, for every player $i$ and every computation-based strategy $g_i$ available to $i$, $\kappa_M(f_i, g_i) \in \Gamma_i$ implies $\kappa_M(f) \in \Gamma_i$.

The concepts of Nash equilibrium in run-based strategies and Nash equilibrium in trace-based strategies are defined analogously, where, importantly, $f_i$ and $g_i$ are required to be of the same type, that is, both run-based or both trace-based. If $\kappa(f) \notin \Gamma_i$ whereas $\kappa(f_i, g_i) \in \Gamma_i$, we also say that player $i$ would like to deviate from $f_i$ to $g_i$. Thus, a run-based profile $f$ is a run-based equilibrium whenever no player would like to deviate from it to some run-based strategy different from $f_i$. Similarly, a trace-based profile $f$ is a trace-based equilibrium if no player would prefer to deviate to another trace-based strategy.

We furthermore say that a computation $\kappa$, run $\rho$, or a trace $\tau$ is sustained by a Nash equilibrium $f = (f_1, \ldots, f_n)$ (of any type) whenever $\kappa = \kappa(f)$, $\rho = \rho(f)$, and $\tau = \tau(f)$, respectively. We also refer to a computation, run, or trace that is sustained by a Nash equilibrium as an equilibrium computation, equilibrium run, and equilibrium trace, respectively.
Computation-based equilibrium is a weaker notion than run-based equilibrium, in the sense that, if $f$ is a run-based equilibrium, there is also a corresponding computation-based equilibrium, but not necessarily the other way round. Run-based equilibrium, in turn, is in a similar way a weaker concept than trace-based equilibrium.

4 Invariance of Nash Equilibria under Bisimilarity

From a computational point of view, one may expect games based on bisimilar concurrent game structures and with identical players’ preferences to display very similar behaviours, in particular with respect to their Nash equilibria. We find that while this is indeed the case for games with computation-based strategies as well as for games with trace-based strategies, for games with run-based strategies the situation is considerably more complicated. A key observation is that, by contrast to computation-based and trace-based strategies, there need not be a natural one-to-one mapping between the sets of run-based strategies in bisimilar concurrent game models. By restricting to so-called bisimulation-invariant run-based strategies, however, we find that order can be restored.

4.1 Computation-based and Trace-based Strategies

If strategies are computation-based, players can have their actions depend on virtually all information that is available in the system. In an important sense full transparency prevails and different actions can be chosen on bisimilar states provided that the computations that led to them are different. As, moreover, the strategies available to player in bisimilar concurrent game structures are identical, we obtain our first main preservation result.

▶ Theorem 1. Let $G = (M, \Gamma_1, \ldots, \Gamma_n)$ and $G' = (M', \Gamma_1, \ldots, \Gamma_n)$ be games on bisimilar concurrent game structures $M$ and $M'$, respectively, and let $f = (f_1, \ldots, f_n)$ be a computation-based profile. Then, $f$ is a Nash equilibrium in computation-based strategies in $G$ if and only if $f$ is a Nash equilibrium in computation-based strategies in $G'$.

Since the sets of traces of two bisimilar concurrent game structure coincide, we can also directly compare their trace-based Nash-equilibria. We find that trace-based Nash equilibria are likewise preserved in CGS-games based on bisimilar concurrent game structures.

▶ Theorem 2. Let $G = (M, \Gamma_1, \ldots, \Gamma_n)$ and $G' = (M', \Gamma_1, \ldots, \Gamma_n)$ be games on bisimilar concurrent game structures $M$ and $M'$, respectively, and $f = (f_1, \ldots, f_n)$ be a trace-based strategy profile. Then, $f$ is a Nash equilibrium in trace-based strategies in $G$ if and only if $f$ is a Nash equilibrium in trace-based strategies in $G'$.

As an immediate consequence of Theorems 1 and 2, we also find that the properties of computations and traces being sustained by, respectively, a computation-based equilibrium and a trace-based equilibrium are preserved under bisimulation.

4.2 Run-based Strategies

The positive results obtained using computation-based and trace-based strategies (with respect to both computation-based goals and trace-based goals) are now followed by a negative result, already mentioned in the introduction of the paper, which establishes that Nash equilibria in run-based strategies—probably the most widely-used strategy model in logic, computer science, and AI—are not preserved by bisimilarity. Previously we observed that the players’ run-based strategies cannot straightforwardly be identified across two
different but bisimilar concurrent game structures. We would therefore have to establish a correspondence between the run-based strategies of different games in an arguably ad hoc way. To cut this Gordian knot, we therefore show the stronger result that the very existence of run-based equilibria is not preserved under bisimilarity. That is, there can be two CGS-games, $G$ and $G'$, that are based on bisimilar concurrent game structures but that $G$ possesses a Nash equilibrium and $G'$ does not.

\textbf{Theorem 3.} The existence of run-based Nash equilibria is not preserved under bisimilarity. That is, there are games $(M, \Gamma_1, \ldots, \Gamma_n)$ and $(M', \Gamma'_1, \ldots, \Gamma'_n)$ on bisimilar concurrent game structures $M$ and $M'$ such that a run-based equilibrium exists in $G$ but not in $G'$.

To see that the above statement holds, recall the three-player game $G_0$ on the concurrent game structure $M_0$ in Figure 1. Assume, as before, that player 1’s goal set $\Gamma_1$ is given by those computations $k$ such that $\tau_{M_0}(k) = v_0, v_1, v_2, \ldots$, implies $p \in v_k$ for any meaningful deviation from $f$.

Accordingly, player 3’s run-based strategy for players 1 and 2 such that $\hat{f}_1(s_0) = f_2(s_0) = a$. Let, furthermore, player 3’s run-based strategy $f_2$ be such that $f_2(s_0) = a$, $f_2(s_0, s_1) = a'$, and $f_2(s_0, s_1') = b$. Let $f^* = (f_1^*, f_2^*, f_3^*)$ and observe that $\rho_M(f^*) = s_0, s_4, s_4, s_4 \ldots$. Accordingly, player 3 has her goal achieved and does not want to deviate from $f^*$. Players 1 and 2 do not have their goals achieved, but do not want to deviate from $f^*$ either. To see this, let $g_1$ be any run-based strategy for 1 such that $g_1(s_0) = b$; observe that this is required for any meaningful deviation from $f^*$ by 1. Then $\rho_M(g_1, f_2^*, f_3^*) = s_0, s_1, s_3, s_3, s_3 \ldots$ or $\rho_M(g_1, f_2^*, f_3^*) = s_0, s_1, s_4, s_4, s_4, \ldots$, depending on whether $f_2(s_0, s_1) = b$ or $f_2(s_0, s_1) = a$, respectively. In either case, player 1 does not get her goal achieved and it follows that she does not want to deviate from $f^*$. An analogous argument—notice that the roles of player 1 and 2 are symmetric—shows that player 2 does not want to deviate from $f^*$ either. We may thus conclude that $f^*$ is a run-based equilibrium in $G_0$.

Now, consider the game $G_1$ on concurrent game structure $M_1$ in Figure 2 with the players’ preferences as in $M_0$. It is easy to check that $M_0$ and $M_1$ are bisimilar. Still, there is no run-based equilibrium in $G_1$. To see this, consider an arbitrary run-based strategy profile $f = (f_1, f_2, f_3)$. First, assume that $\rho_M(f) = s_0, s_1, s_2, s_2, s_2, \ldots$. Then, player 1 gets his goal achieved and players 2 and 3 do not. If $f_1(s_0, s_1) = a$ then $f_3(s_0, s_1) = b$ and player 3 would like to deviate and play a strategy $g_3$ with $g_3(s_0, s_1) = a$. On the other hand, if $f_1(s_0, s_1) = b$, player 3 would like to deviate and play a strategy $g_3$ with $g_3(s_0, s_1) = b$. Player 3 would similarly like to deviate from $f$ if we assume that $\rho_M(f) = s_0, s_1, s_3, s_3, s_3 \ldots$, in whose case it is player 2 who gets his goal achieved. Now, assume that $\rho_M(f) = s_0, s_1, s_4, s_4, s_4 \ldots$. In this case player 3 does get her goal achieved, but players 1 and 2 do not. However, player 1 would like to deviate to a strategy $g_1$ with $g_1(s_0, s_1) = b$ or $g_1(s_0, s_1) = a$, depending on whether $f_3(s_0, s_1) = a$ or $f_3(s_0, s_1) = b$; in a similar fashion, player 2 would like to deviate to a strategy $g_2$ with $g_2(s_0, s_1) = b$ if $f_1(s_0, s_1) = a'$, and to one with $g_2(s_0, s_1) = a$ if $f_1(s_0, s_1) = b'$. Finally, assume that $\rho_M(f) = s_0, s_4, s_4, s_4, \ldots$. Then, neither player 1 nor player 2 gets his goal achieved. Now either $f_3(s_0, s_1) \in \{a, b\}$ or $f_3(s_0, s_1) \in \{a', b'\}$. If the former, player 1 would like to deviate to a strategy $g_1$ with $g_1(s_0) \neq f_1(s_0)$ and $g_1(s_0, s_1) \neq f_3(s_0, s_1)$. If the latter, player 2 would like to deviate to a strategy $g_2$ with $g_2(s_0) \neq f_2(s_0)$ and either $g_2(s_0, s_1) = b$ or $f_3(s_0, s_1) = a'$ or $f_3(s_0, s_1) = a$ if $f_3(s_0, s_1) = b'$. We can then conclude that the CGS-game $G_1$ does not have any run-based Nash equilibria.
The main idea behind this counter-example is that in $G_0$ player 3 can distinguish which player deviates from $f^*$ if the state reached after the first round is not $s_4'$: if that state is $s_1$, player 1 deviated, otherwise player 2 did. By choosing either $a'$ or $b'$ at $s_1$, and either $a$ or $b$ at $s_4'$, player 3 can guarantee that neither player 1 nor player 2 gets his goal achieved (“punish” them) and thus deter their deviating from $f^*$. This possibility to punish deviations from $f^*$ by players 1 and 2 in a single strategy is not available in $G_1$: choosing from $a$ and $b$ will induce a deviation by player 1, choosing from $a'$ and $b'$ one by player 2.

Observe that the games $G_0$ and $G_1$ do not constitute a counter-example against the preservation under bisimilarity of either computation-based equilibria or trace-based equilibria. The reasons why such games fail to do so, however, are distinct. For the setting of computation-based strategies, player 3 can still distinguish and “punish” the deviating player in $G_1$ as $(a, b, a)$ and $(b, a, a)$ are different directions and player 3 can still have his action at $s_1$ depend on which of these is played at $s_0$. By contrast, if we assume trace-based strategies, player 3 has to choose the same action at both $s_1$ and $s_4'$ in $G_0$. As a consequence, and contrarily to computation-based equilibria, trace-based equilibria exist in neither $G_0$ nor $G_1$.\footnote{Based on a similar example, Almagor et al. \cite{1} also observe that the existence of Nash equilibria in a CGS-game may depend on the type of strategy that is being considered. Note, however, that this is quite different from our observation that Nash equilibria may differ in different but bisimilar CGS-games given a fixed type of strategy, viz., run-based strategies.}

Also note that, as the goal sets $\Gamma_1, \Gamma_2,$ and $\Gamma_3$ are run-based as well as computation-based both in $G_1$ and $G_2$, the counter-example still holds if preferences are more fine-grained.

Observe at this point that $s_3$ and $s_4'$ are bisimilar states. Yet, players are allowed to have run-based strategies (which depend on state histories only) that choose different actions at bisimilar states. The above counter-example shows how this relative richness of strategies makes a crucial difference. This raises the question as to whether we actually want players to adopt run-based strategies in which they choose different actions at bisimilar states. This observation leads us to the next section.

### 4.3 Bisimulation-invariant Run-based Strategies

Bisimilarity formally captures an informal concept of observational indistinguishability on the part of an external observer of the system. The players in a concurrent game structure are in essentially the same situation as an external observer, if they are assumed to be only able to observe the behaviour of the other players, without knowing their internal makeup.

Drawing on this idea of indistinguishability, it is natural to assume that players cannot distinguish statewise bisimilar runs and, as a consequence, can only adopt strategies that choose the same action at runs that are statewise bisimilar. The situation is comparable to the one in extensive games of imperfect information, in which players are required to choose the same action in histories that are in the same information set (cf., for instance, \cite{22, 17}).

To make this idea precise, we say that a run-based strategy $f_i$ is bisimulation-invariant if $f_i(\pi) = f_i(\pi')$ for all histories $\pi$ and $\pi'$ that are statewise bisimilar. The concept of Nash equilibrium is then similarly restricted to bisimulation-invariant strategies. Formally, a profile $f = (f_1, \ldots, f_n)$ of bisimulation-invariant strategies is a Nash equilibrium in bisimulation-invariant strategies (or a bisimulation-invariant equilibrium) in a game $(M, \Gamma_1, \ldots, \Gamma_n)$ if for all players $i$ and every bisimulation-invariant strategy $g_i$ for $i$, $\tau(f_{-i}, g_i) \in \Gamma_i$ implies $\tau(f) \in \Gamma_i$. In contrast to the situation for general run-based strategies, we find that computations and traces that are sustained by a bisimulation-invariant Nash equilibrium are preserved by bisimulation. Let $M$ and $M'$ be bisimilar concurrent game structures.
Based on the concept of statewise bisimilarity, we associate with every bisimulation-invariant strategy \( f_i \) for player \( i \) in \( M \), another bisimulation-invariant strategy \( \tilde{f}_i \) for player \( i \) in \( M' \) such that for all \( \pi \in \text{runs}_M \) and \( a \in A_i \), we have \( \tilde{f}_i(\pi) = a \) if \( f_i(\pi') = a \) for some \( \pi' \in \text{runs}_{M'} \) with \( \pi \not\sim \pi' \). Transitivity of \( \not\sim \) guarantees that \( \tilde{f}_i \) is well-defined. Given a profile \( f = (f_1, \ldots, f_n) \), let \( \tilde{f} = (\tilde{f}_1, \ldots, \tilde{f}_n) \). It can then be shown that \( f \) and \( \tilde{f} \) induce identical computations, that is, \( \kappa_M(f) = \kappa_M(\tilde{f}) \), which prepares the ground for the following preservation result.

**Theorem 4.** Let \( G = (M, \Gamma_1, \ldots, \Gamma_n) \) and \( G' = (M', \Gamma_1, \ldots, \Gamma_n) \) be games on bisimilar concurrent game structures \( M \) and \( M' \), respectively. Then, profile \( f \) is a bisimulation-invariant equilibrium in \( G \) if and only if \( \tilde{f} \) is a bisimulation-invariant equilibrium in \( G' \).

As an immediate corollary of Theorem 4, it follows that the property of a computation or trace to be sustained by a bisimulation-invariant equilibria is also preserved under bisimilarity.

### 4.4 Two-player Games with Run-based Strategies

We now consider the setting of two-player games with run-based strategies and trace-based goals (which include temporal logic goals). This is an important special case, since run-based strategies, as we emphasised in the introduction, are the conventional model of strategies used in logics such as ATL* or Strategy Logic (SL), as well as in multi-agent systems represented as concurrent game structures [4, 19].

The counterexample against the preservation of existence of Nash equilibria in Section 4.2 involved three players. We find that, if preferences are computation-based, the example can be adapted so as to involve only two players. Hence, we have the following result.

**Theorem 5.** There are two-player games \( (M, \Gamma_1, \Gamma_2) \) and \( (M', \Gamma_1, \Gamma_2) \) on bisimilar concurrent game structures \( M \) and \( M' \) with \( \Gamma_1 \) and \( \Gamma_2 \) computation-based such that a run-based Nash equilibrium exists in \( G \) but not in \( G' \).

By contrast, if players' preferences are required to be trace-based in any two models to be compared, sustenance of traces by run-based equilibrium—and hence also the existence of Nash equilibria—is preserved under bisimilarity. To establish this, we associate with each run-based profile \( f = (f_1, f_2) \) a bisimulation-invariant profile \( \tilde{f} = (\tilde{f}_1, \tilde{f}_2) \). We then show that \( f \) and \( \tilde{f} \) induce the same traces and that \( \tilde{f} \) is a bisimulation invariant equilibrium, if \( f \) is a run-based equilibrium. We can then leverage Theorem 4, which yields the result.

We only give the definition of \( \tilde{f}_1 \), as the construction of \( \tilde{f}_2 \) is analogous. The key idea is to select for every finite run \( \pi = s_0, \ldots, s_k \) a unique representative \( \tilde{\pi}^{s_0, \ldots, s_k} \) of the equivalence class \([\pi]_\prec \) of runs statewise bisimilar to \( \pi \) and then define \( \tilde{f}_1 \) such that \( \tilde{f}_1(\pi) = f_1(\tilde{\pi}^{s_0, \ldots, s_k}) \). This ensures that \( \tilde{f}_1 \) is bisimulation-invariant. The representative \( \tilde{\pi}^{s_0, \ldots, s_k} \), however, has to be chosen carefully. Assuming that from every subset of actions we can always select a least element, we define inductively for every finite run \( \pi = s_0, \ldots, s_k \) in \( \text{runs}_M \) another statewise bisimilar finite run \( \tilde{\pi}^{s_0, \ldots, s_k} = t_0, \ldots, t_k \) in \( \text{runs}_M \) such that \( t_0 = s_0^\pi \) and, for every \( 0 \leq m < k \), \( t_{m+1} = \delta(t_m, x_1, x_2) \), where \( x_1 \) and \( x_2 \) are actions available to players 1 and 2 at \( t_m \) such that:

- \( x_1 = f_1(t_0, \ldots, t_m) \) and \( x_2 = f_2(t_0, \ldots, t_m) \), if \( \delta(t_m, f_1(t_0, \ldots, t_m), f_2(t_0, \ldots, t_m)) \sim s_{m+1} \),
- \( x_1 = f_1(t_0, \ldots, t_m) \) and \( x_2 \) is the least action available to player 2 at \( t_m \) such that \( \delta(t_m, f_1(t_0, \ldots, t_m), x_2) \sim s_{m+1} \), if \( \delta(t_m, f_1(t_0, \ldots, t_m), f_2(t_0, \ldots, t_m)) \not\sim s_{m+1} \) and \( x_2 \) exists, and
- \( (x_1, x_2) \) is lexicographically the least pair of actions available to players 1 and 2 at \( t_m \) such that \( \delta(t_m, x_1, x_2) \sim s_{m+1} \), otherwise.
By means of an inductive argument it can easily be shown that \( s_0, \ldots, s_k \sim \pi^{s_0, \ldots, s_k} \), which suffices for \( \pi^{s_0, \ldots, s_k} \) to be well-defined, that is, that \( x_1 \) and \( x_2 \) exist for every \( 0 \leq m < k \). As, \( s_0, \ldots, s_k \sim t_0, \ldots, t_k \) implies \( \pi^{s_0, \ldots, s_k} = \pi^{t_0, \ldots, t_k} \), it follows that \( f_1 \) is properly defined as a bisimulation-invariant strategy. Importantly, \( \rho_{M}(f_1, f_2) = \rho_{M}(f_1, f_2) \), and hence also \( \tau_{M}(f_1, f_2) = \tau_{M}(f_1, f_2) \). Moreover, we find by a non-trivial argument that, if \( f = (f_1, f_2) \) is a run-based equilibrium, so is \( \hat{f} = (f_1, f_2) \). Finally, as a bisimulation-invariant profile is a bisimulation-invariant equilibrium if and only if it is a run-based equilibrium, we are in a position to leverage Theorem 4 and obtain our result.

**Theorem 6.** Let \( G = (M, \Gamma_1, \Gamma_2) \) and \( G' = (M', \Gamma_1, \Gamma_2) \) be two two-player games such that \( \Gamma_1 \) and \( \Gamma_2 \) are trace-based in both \( M \) and \( M' \) and let \( \tau \) be a trace in \( traces_M \). Then, \( \tau \) is sustained by a run-based equilibrium in \( G \) if and only if \( \tau \) is sustained by a run-based equilibrium in \( G' \).

As an immediate consequence of Theorem 6 we also find that the existence of Nash equilibria is also preserved in two-player games with trace-based preferences. The case in which players’ preferences are run-based, however, we have to leave as an open question.

## 5 Strategy Logics: New Semantic Foundations

Several logics for strategic reasoning have been proposed in the literature of computer science and AI, such as ATL* [4], Strategy Logic [19, 8], Coalition Logic [23], Coordination Logic [12], Game Logic [24], and Equilibrium Logic [15]. In several cases, the model of strategies that is used is the one that we refer to as run-based in this paper, that is, strategies are functions from finite sequences of states (of some arena) to actions/decisions/choices of players in a given game. As can be seen from our results so far, of the four options we have explored, run-based strategies form the least desirable model of strategies from a semantic point of view since in such a case Nash equilibrium is not preserved under bisimilarity.

This does not necessarily immediately imply that a particular logic with a run-based strategy model is not invariant under bisimilarity. For instance, ATL* is a bisimulation-invariant logic and, as shown in [13] one can reason about Nash equilibrium using ATL* only up-to bisimilarity. A question then remains: whether any of these logics for strategic reasoning becomes invariant under bisimilarity—as explained before, a desirable property—if one changes the model of strategies considered there to, for instance, computation-based or trace-based strategies. We find that this question has a satisfactory positive answer in some cases. In particular, we will consider the above question in the context of Strategy Logic as studied in [19], and in doing so we will provide new semantic foundations for strategy logics.

Let us start by introducing the syntax and semantics under the run-based model of strategies for Strategy Logic (SL [19]) as it has been given in [20]. Syntactically, SL extends LTL with two strategy quantifiers, \( (\langle x \rangle \varphi) \) and \( ([x] \varphi) \), and an agent binding operator \( (i, x) \), where \( i \) is an agent and \( x \) is a variable. These operators can be read as “there exists a strategy \( x^* \), “for all strategies \( x^* \)”, and “bind agent \( i \) to the strategy associated with the variable \( x^* \)”, respectively. SL formulae are inductively built from a set of atomic propositions \( AP \), variables \( Var \), and agents \( Ag \), using the following grammar, where \( p \in AP, x \in Var \), and \( i \in Ag \):

\[
\varphi ::= p \mid \neg \varphi \mid \varphi \land \varphi \mid X \varphi \mid \varphi U \varphi \mid \langle (x) \rangle \varphi \mid [x] \varphi \mid (i, x) \varphi.
\]

We can now present the semantics of SL formulae, where \( Str \) denotes the set of all strategies of some specified type, i.e., computation-based, run-based, or trace-based. Given a concurrent game structure \( M \), for all SL formulae \( \varphi \), states \( s \in St \) in \( M \), and assignments
\[ \chi \in \text{Asg} = (\text{Var} \cup \text{Ag}) \rightarrow \text{Str}, \text{ mapping variables and agents to strategies, the relation } M, \chi, s \models \varphi \text{ is defined as follows:} \]

1. For the Boolean and temporal cases, the semantics is standard (see, e.g., [19]:
   a. \[ M, \chi, s \models \langle \langle x \rangle \rangle \varphi \text{ if there is a strategy } f \in \text{Str} \text{ such that } M, \chi[x \mapsto f], s \models \varphi; \]
   b. \[ M, \chi, s \models [[x]] \varphi \text{ if for all strategies } f \in \text{Str} \text{ we have that } M, \chi[x \mapsto f], s \models \varphi. \]

2. For all formulae \( \varphi \) and variables \( x \in \text{Var} \) we have:
   a. \[ M, \chi, s \models \langle \langle x \rangle \rangle \varphi \text{ if there is a strategy } f \in \text{Str} \text{ such that } M, \chi[x \mapsto f], s \models \varphi; \]
   b. \[ M, \chi, s \models [[x]] \varphi \text{ if for all strategies } f \in \text{Str} \text{ we have that } M, \chi[x \mapsto f], s \models \varphi. \]

3. For all \( i \in \text{Ag} \) and \( x \in \text{Var} \) we have \[ M, \chi, s \models (i, x) \varphi \text{ if } M, \chi[i \mapsto \chi(x)], s \models \varphi. \]

Intuitively, rules 2a and 2b, respectively, are used to interpret the existential \( \langle \langle x \rangle \rangle \) and universal \( [[x]] \) quantifiers over strategies, and rule 3 is used to bind an agent to the strategies associated with variable \( x \). All other rules are as in LTL over concurrent game structures. Moreover, for a sentence \( \varphi \), i.e., a formula with no free variables or agents [19], we say that \( M \) satisfies \( \varphi \), and write \( M \models \varphi \), where \( \emptyset \) is the empty assignment.

As it can be seen from its semantics, SL can be interpreted under different models of strategies and goals. As it was originally formulated, SL considers run-based strategies and trace-based preferences/goals [21]. More specifically, the model of goals is a proper subset of the trace-based one, represented by LTL goals over the set \( \text{AP} \) of variables. Let us fix an interpretation of strategies. We say that SL with that kind of model is invariant under bisimulation if, for all sentences \( \varphi \) and bisimilar models \( M_1 \) and \( M_2 \), it holds that \( M_1 \models \varphi \) iff \( M_2 \models \varphi \). In SL, it is possible to represent the existence of a Nash equilibrium in a concurrent game structure [19]. This implies, given Theorem 3, that SL under the standard (run-based model) interpretation is not invariant under bisimulation, as the formula expressing the existence of a Nash equilibrium can distinguish between two bisimilar models.

We now consider SL under the model of computation-based strategies, and find that in such a case SL becomes invariant under bisimilarity. Formally, we have the following result.

\[ \text{Theorem 7. SL with the computation-based model of strategies is invariant under bisimilarity.} \]

An analogous statement to the above Theorem can also be proved if we consider the model of trace-based strategies, leading to the next result on the semantics of SL.

\[ \text{Theorem 8. SL with the trace-based model of strategies is invariant under bisimilarity.} \]

### 6 Concluding Remarks and Related Work

We have shown that in the conventional model of strategies used in logic, computer science, and AI, the existence of a given Nash equilibrium is not preserved under bisimilarity, in particular for multi-player games played over concurrent game structures. With a few examples we also illustrated some of the adverse implications of this result, for instance, in the context of automated formal verification. To resolve this problem, we furthermore investigated alternative models of strategies which exhibit some desirable properties, in particular, allowing for a formalisation of Nash equilibrium that is invariant under bisimilarity.

We studied applications of these models and found that through their use, not only Nash equilibria becomes invariant under bisimilarity, but also full logics such as Strategy Logic [21, 8, 19]. This renders it possible to combine commonly used optimisation techniques for model checking with decision procedures for the analysis of Nash equilibria, thus overcoming a critical problem of this kind of logics regarding practical applications via automated verification. Some work also in the intersection between bisimulation equivalences, concurrent game structures, and Nash equilibria is summarised next.

---

CONCUR 2017
Nash Equilibrium and Bisimulation Invariance

Logics for Strategic Reasoning.

From the current (enormous) set of logics for strategic reasoning in the literature, ATL* [4] and SL [19] stand out, both due to their adoption by a number of practical tools for automated verification and because of their expressive power. On the one hand, ATL* is known to be invariant under bisimilarity using the conventional model of strategies. As such, Nash equilibria can be expressed within ATL* only up to bisimilarity [13]. On the other hand, SL, which is strictly more expressive than ATL*, allows for a simple specification of Nash equilibria, but suffers from not being invariant under bisimilarity with respect to the conventional model of strategies. In this paper, we have put forward a number of solutions to this problem. An additional advantage of replacing the model of strategies for SL (and therefore for concurrent game structures) is that other solution concepts in game theory also become invariant under bisimilarity. For instance, subgame-perfect Nash equilibria and strong Nash equilibria—which are widely used when considering, respectively, dynamic behaviour and cooperative behaviour in multi-agent systems—can also be expressed in SL. Our results therefore imply that these concepts are also invariant under bisimilarity, when considering games over concurrent game structures and goals given by LTL formulae.

Bisimulation Equivalences for Multi-Agent Systems.

Even though bisimilarity is probably the most widely used behavioural equivalence in concurrency, in the context of multi-agent systems other relations may be preferred, for instance, equivalence relations that take a detailed account of the independent interactions and behaviour of individual components in a multi-agent system. In such a setting, “alternating” relations with natural ATL* characterisations have been studied [3, 11]. Our results also apply to such alternating relations. On the one hand, the counter-example shown in Figures 1 and 2 also apply to such alternating (bisimulation) relations. On the other hand, because these alternating relations can be characterised in ATL*, they are at most as strong as bisimilarity. These two facts together imply that Nash equilibria is not preserved by the alternating (bisimulation) equivalence relations in [3, 11] either. Nevertheless, as discussed in [27], the right notion of equivalence for games and their game theoretic solution concepts is most certainly an important and interesting topic of debate.

Computations vs Traces.

A difference between computations and traces is that even though Nash equilibria and their existence are preserved under bisimilarity by three of the four strategy models we have studied, it is not the case that with each strategy model we obtain the same set of Nash equilibria in a given system, or that we can sustain the same set of computations or traces. For instance, consider the games in Figures 1 and 2. As we discussed above, if we consider the model of computation-based strategies and LTL goals (i.e., trace-based goals) as shown in the example, then we obtain two games, each with an associated non-empty set of Nash equilibria, which is preserved by bisimilarity. However, if we consider, instead, the model of trace-based strategies and same LTL goals, then we obtain two games both with empty sets of Nash equilibria—thus, in this case, the non-existence of Nash equilibria is preserved by bisimilarity! To observe this, note that whereas in the case of computation-based strategies player 3 can implement a uniform “punishment” strategy for both player 1 and player 2, in the case of trace-based strategies player 3 cannot do so, even in the game in Figure 1.
Two-Player Games with Trace-Based Goals.

In this paper we also showed that if we consider two-player games together with the conventional model of strategies, the problems that arise with respect to the preservation of Nash equilibria disappear. This is indeed an important finding since most verification games (e.g., model and module checking, synthesis, etc.) can be phrased in terms of two-player games together with temporal logic specifications (e.g., using LTL, CTL, or ATL*). Our results, then, provide proof that, if only two-player games and temporal logic goals are needed, then all equilibrium analyses can be carried out using the conventional model of strategies—along with their associated reasoning tools and verification techniques.

Boolean Game Structures.

Boolean game structures are the special type of concurrent game structure in which each player $i$ has unique control over a subset $\text{AP}_i$ of propositional variables and the set $\text{Ac}_i(s)$ of actions available to player $i$ at state $s$ is a non-empty subset of partial valuations in $2^{\text{AP}_i}$. The key idea is that the choice player $i$ makes with respect to the variables in $\text{AP}_i$ at a state $s$ defines the labelling of the subsequent state with respect to $\text{AP}_i$. Formally, for every direction $d' = (a_1, \ldots, a_n)$ in $2^{\text{AP}_1} \times \cdots \times 2^{\text{AP}_n}$ and every state $s$, it holds that $\delta(s, d') = s'$ implies $\lambda(s') = a_1 \cup \cdots \cup a_n$.

Boolean game structures are a particularly well-behaved class of games, of which iterated Boolean games [14] and multi-agent systems modelled using the Reactive Modules specification language [2] are special cases. We thus find that, in Boolean game structures, (existence of) Nash equilibrium is invariant under bisimilarity regardless of the model of strategies or goals that one chooses. As in this setting, it is readily shown that if two finite runs are statewise bisimilar, they have to be identical, it immediately follows that run-based strategies are bisimulation-invariant. Hence, by Theorem 4 that (the existence of) run-based equilibria is invariant under bisimulation. For the settings with computation-based and trace-based strategies, Theorems 1 and 2 give the result, respectively.

References

Nash Equilibrium and Bisimulation Invariance


Goal-Driven Unfolding of Petri Nets

Thomas Chatain\textsuperscript{1} and Loïc Paulevé\textsuperscript{2}

\textbf{Abstract}

Unfoldings provide an efficient way to avoid the state-space explosion due to interleavings of concurrent transitions when exploring the runs of a Petri net. The theory of adequate orders allows one to define finite prefixes of unfoldings which contain all the reachable markings. In this paper we are interested in reachability of a single given marking, called the goal. We propose an algorithm for computing a finite prefix of the unfolding of a 1-safe Petri net that preserves all minimal configurations reaching this goal. Our algorithm combines the unfolding technique with on-the-fly model reduction by static analysis aiming at avoiding the exploration of branches which are not needed for reaching the goal. We present some experimental results.

\textbf{1998 ACM Subject Classification} F.1.1 Models of Computation, F.1.2 Modes of Computation, D.2.4 Software/Program Verification, J.3 Life and Medical Sciences

\textbf{Keywords and phrases} model reduction, reachability; concurrency, unfoldings, Petri nets, automata networks

\textbf{Digital Object Identifier} 10.4230/LIPIcs.CONCUR.2017.18

\section{Introduction}

Analysing the possible dynamics of a concurrent system expressed as Petri nets can be eased by means of unfoldings and their prefixes which avoid exploring redundant interleaving of transitions.

In this paper, we propose a method which combines the unfolding technique with model reduction in order to explore efficiently and completely the minimal configurations (partially ordered occurrences of transitions) which lead to a given goal marking/marked place. In particular, we aim at ignoring configurations that cannot reach the goal, but also configurations containing transient cycles.

The goal-driven unfolding relies on calling, on the fly, an external model reduction procedure which identifies transitions not part of any minimal configuration for the goal reachability from the current marking. Those useless transitions are then skipped by the unfolding.

We show how model reduction can be applied to the unfolding of a safe Petri net $\mathcal{N}$ in such a way that it preserves minimal configurations. Then we present an algorithm to construct a corresponding goal-driven finite prefix.

We illustrate this procedure on the Petri net of Figure 1. The goal is $\{p'_3, p_5\}$. Notice that only one occurrence of $t_3$ is needed to reach the goal. So, after the corresponding event, $t_3$ can be declared useless. Also, after firing $t_1$, $t'_2$ is fireable but firing it makes the goal
unreachable. Therefore, a reduction procedure may declare that $t'_2$ is useless once $t_1$ has occurred, allowing one to avoid exploring this branch. Symmetrically, $t'_1$ is useless once $t_2$ has occurred. It is easy to imagine a larger model where a large piece of behaviour would be reachable from $\{p_1, p'_2, p_3\}$ (but would not allow to reach the goal); or from $\{p_3, p_4\}$ (but would involve transient cycles): the usual complete finite prefix would explore such configurations, while our model reduction can avoid their computation.

The design of the model reduction procedure which identifies useless transitions is out of the scope of the paper. Instead, we consider it as a blackbox, and design our approach assuming the reduction preserves all the minimal (acyclic) sequences of transitions leading to the goal. Moreover, to be of practical interest, the reduction should show a complexity lower than the reachability problem (PSPACE-complete [7]).

As detailed in Section 4.2, skipping transitions declared useless by a reduction procedure involves non-trivial modifications to the algorithm for computing the prefix of the unfolding. Indeed, a particular treatment of cut-offs has to be introduced in order to ensure that the resulting goal-driven prefix includes all the minimal sequences of transitions.

The goal-driven unfolding has practical applications in systems biology [19]. Indeed, numerous dynamical properties relevant for biological networks focus on the reachability of the activity of a particular node in the network, typically a transcription factor known to control a given cellular phenotype. In this perspective, having computational methods that can be tailored for such narrow reachability properties is of practical interest. The completeness of the minimal sequences of transitions for the goal reachability is critical for several analyses of biological system dynamics. An example is the identification of parts of the network that play a central role to activate a node of interest. By altering such parts (e.g., with mutations) one can expect prevent such an activation [18]. If the analysis considers
only a partial set of minimal sequences, there is no guarantee that the predicted mutations are sufficient to prevent the goal reachability.

We illustrate the benefit of the goal-driven unfolding on models of biological networks by instantiating our method with a reduction procedure introduced in [17].

**Related Work**

Numerous work address the computation of reachable states in concurrent systems using unfoldings. Several algorithms for checking reachability based on a previously computed finite complete prefix of a Petri net are compared in [12]. Over-approximations of the unfolding (i.e., which contains all the reachable markings, but potentially more) for graph transformation systems are defined in [2].

Despite the negative result [9] which states that depth-first-search strategies are not correct for classical unfolding algorithms, [3] defines directed unfolding of Petri nets, which is closely related to our goal-driven unfolding. They rely on a heuristic function (on configurations) to generate an ordering of the events for making a given transition appear as soon as possible during the unfolding. In addition, they can consider heuristic functions to detect configuration from which the goal transition is not reachable. In such a case, no extension will be made to that configuration, which may significantly prune the computed prefix. The major difference with the work presented in the paper is that directed unfolding does not prune transitions leading to spurious transient cycles on the way to the goal. Actually, in their terms, our reduction procedure would not be considered safely pruning because we discard (non-minimal) configurations reaching the goal. In a sense, the reduction they achieve on the prefix size corresponds to the extreme case when our external reduction procedure returns the full model if the goal is reachable, and the empty model if not. Indeed, except for the case when the goal is detected as non-reachable, all the other configurations are kept in the directed unfolding, whereas our approach can potentially output a prefix containing only, but all, minimal configurations for the goal reachability.

Less related to our work, static analysis techniques were also used in combination with partial order reductions: [13, 21] rely on an on-the-fly detection of independence relations by static analysis, to improve partial order reductions.

**Outline**

Section 2 gives the basics of Petri net unfoldings and of their complete finite prefixes. The concepts of minimal configuration and model reduction are introduced in Section 3, and Section 4 details the goal-driven unfolding and prefix with proofs of completeness. Finally, Section 5 applies the goal-driven prefix to actual biological models, and Section 6 concludes this paper.

**2 Unfoldings of Petri nets**

In this section, we explain the basics of Petri net unfoldings. A more extensive treatment of the theory explained here can be found, e.g., in [8]. Roughly speaking, the unfolding of a Petri net $\mathcal{N}$ is an “acyclic” Petri net $\mathcal{U}$ that has the same behaviours as $\mathcal{N}$ (modulo homomorphism). In general, $\mathcal{U}$ is an infinite net, but if $\mathcal{N}$ is safe, then it is possible [16] to compute a finite prefix $\mathcal{P}$ of $\mathcal{U}$ that is “complete” in the sense that every reachable marking of $\mathcal{N}$ has a reachable counterpart in $\mathcal{P}$. Thus, $\mathcal{P}$ represents the set of reachable markings of $\mathcal{N}$. Figure 1 shows a Petri net and a finite complete prefix of its unfolding.
We now give some technical definitions to introduce unfoldings formally.

**Definition 1 (Safe Petri Net).** A *safe Petri net* is a tuple $\mathcal{N} = (\mathcal{P}, \mathcal{T}, \mathcal{F}, \mathcal{M}_0)$ where $\mathcal{P}$ and $\mathcal{T}$ are sets of nodes (called places and transitions respectively), and $\mathcal{F} \subseteq (\mathcal{P} \times \mathcal{T}) \cup (\mathcal{T} \times \mathcal{P})$ is a flow relation (whose elements are called arcs). A subset $\mathcal{M} \subseteq \mathcal{P}$ of the places is called a marking, and $\mathcal{M}_0$ is a distinguished initial marking.

For any node $x \in \mathcal{P} \cup \mathcal{T}$, we call pre-set of $x$ the set $\{ y \in \mathcal{P} \cup \mathcal{T} \mid (y, x) \in \mathcal{F} \}$ and post-set of $x$ the set $\{ y \in \mathcal{P} \cup \mathcal{T} \mid (x, y) \in \mathcal{F} \}$. These notations are extended to sets $\mathcal{Y} \subseteq \mathcal{P} \cup \mathcal{T}$, with $\mathcal{Y} \cdot = \cup_{x \in \mathcal{Y}} x$ and $\mathcal{Y}^\ast = \cup_{x \in \mathcal{Y}} x^\ast$.

A transition $t \in \mathcal{T}$ is *enabled* at a marking $\mathcal{M}$ if and only if $t \subseteq \mathcal{M}$. Then $t$ can *fire*, leading to the new marking $\mathcal{M}' = (\mathcal{M} \setminus t)^\ast$ (called a *firing sequence*). A firing sequence is a (finite or infinite) word $w = t_1t_2\ldots$ over $\mathcal{T}$ such that there exist markings $\mathcal{M}_1, \mathcal{M}_2, \ldots$ such that $\mathcal{M}_0 \xrightarrow{t_1} \mathcal{M}_1 \xrightarrow{t_2} \mathcal{M}_2 \ldots$ For any such firing sequence $w$, the markings $\mathcal{M}_1, \mathcal{M}_2, \ldots$ are called reachable markings.

The Petri nets we consider are said to be *safe* because we will assume that any reachable marking $\mathcal{M}$ is such that for any $t \in \mathcal{T}$ that can fire from $\mathcal{M}$ leading to $\mathcal{M}'$, the following property holds: $\forall p \in \mathcal{M} \cap \mathcal{M}', p \in t \cap \mathcal{M}' \forall p \notin t \cup \mathcal{M}'$.

Figure 1 (left) shows an example of a safe Petri net. The places are represented by circles and the transitions by rectangles (each one with a label identifying it). The arrows represent the arcs. The initial marking is represented by dots (or tokens) in the marked places.

**Definition 2 (Causality, conflict, concurrency).** Let $\mathcal{N} = (\mathcal{P}, \mathcal{T}, \mathcal{F}, \mathcal{M}_0)$ be a net and $t, t' \in \mathcal{T}$ two transitions of $\mathcal{N}$. We say that $t$ is a causal predecessor of $t'$, noted $t < t'$, if there exists a non-empty path of arcs from $t$ to $t'$. We note $t \leq t'$ if $t < t'$ or $t = t'$. If $t \leq t'$ or $t' \leq t$, then $t$ and $t'$ are said to be causally related. The set of causal predecessors of $t$ is denoted $[t]$. We write $[t]$ for $[t] \cup \{ t \}$, which we call the causal past of $t$. Transitions $t$ and $t'$ are in *conflict*, noted $t \# t'$, if there exist $u, v \in \mathcal{T}$ such that $u \neq v$, $u \leq t$, $v \leq t'$ and $u \cap t' \neq \emptyset$. We call $t$ and $t'$ *concurrent*, noted $t \& t'$, if they are neither causally related nor in conflict.

As we said before, an unfolding is an “acyclic” net. This notion of acyclicity is captured by Definition 3. As is convention in the unfolding literature, we shall refer to the places of an occurrence net as *conditions* and to its transitions as *events*. Due to the structural constraints, the firing sequences of occurrence nets have special properties: if some condition $c$ is marked during a run, then the token on $c$ was either present initially or produced by one particular event (the single event in $\mathcal{E}$); moreover, once the token on $c$ is consumed, it can never be replaced by another token, due to the acyclicity constraint on $<$.

**Definition 3 (Occurrence net).** An occurrence net $\mathcal{O} = (\mathcal{C}, \mathcal{E}, \mathcal{G}, \mathcal{C}_0)$ is a Petri net $(\mathcal{P}, \mathcal{T}, \mathcal{F}, \mathcal{M}_0)$ with $\mathcal{P} = \mathcal{C}$, $\mathcal{T} = \mathcal{E}$, $\mathcal{F} = \mathcal{G}$, $\mathcal{M}_0 = \mathcal{C}_0$ for which:

1. The causality relation $\prec$ is acyclic;
2. $|\mathcal{C}_0| \leq 1$ for all places $p$, and $p \in \mathcal{M}_0$ iff $|\mathcal{C}_0| = 0$;
3. for every transition $t$, $t \# t$ does not hold, and $\{ x \mid x \preceq t \}$ is finite.

**Definition 4 (Configuration, cut).** Let $\mathcal{O} = (\mathcal{C}, \mathcal{E}, \mathcal{G}, \mathcal{C}_0)$ be an occurrence net. A set $\mathcal{E} \subseteq \mathcal{E}$ is called configuration (or process) of $\mathcal{O}$ if (i) $\mathcal{E}$ is causally closed, i.e. for all $c, c' \in \mathcal{E}$ with $c' < c$, if $c \in \mathcal{E}$ then $c' \in \mathcal{E}$; and (ii) $\mathcal{E}$ is conflict-free, i.e. if $c, c' \in \mathcal{E}$, then $\lnot (c \# c')$. The cut of $\mathcal{E}$, denoted $\text{Cut}(\mathcal{E})$, is the set of conditions $(\mathcal{C}_0 \cup \mathcal{E}^\ast) \setminus \mathcal{E}$.

An occurrence net $\mathcal{O}$ with a net homomorphism $h$ mapping its conditions and events to places and transitions of a net $\mathcal{N}$ is called a branching process of $\mathcal{N}$. Intuitively, a configuration of $\mathcal{O}$ is a set of events that can fire during a firing sequence of $\mathcal{N}$, and its cut is the set of conditions marked after that sequence.
2.1 Unfolding

Let $\mathcal{N} = \langle P, T, F, M_0 \rangle$ be a safe Petri net. The unfolding $U = \langle C, E, G, C_0 \rangle$ of $\mathcal{N}$ is the unique (up to isomorphism) maximal branching process such that the firing sequences and reachable markings of $U$ represent exactly the firing sequences and reachable markings of $\mathcal{N}$ (modulo $h$). $U$ is generally infinite and can be inductively constructed as follows:

1. The conditions $C$ are a subset of $(E \cup \{ \bot \}) \times P$. For a condition $c = \langle x, p \rangle$, we will have $x = \bot$ iff $c \in C_0$; otherwise $x$ is the singleton event in $\cdot c$. Moreover, $h(c) = p$. The initial marking $C_0$ contains one condition $\langle \bot, p \rangle$ per initially marked place $p$ of $\mathcal{N}$.

2. The events $E$ are a subset of $2^C \times T$. More precisely, we have an event $e = \langle C', t \rangle$ for every set $C' \subseteq C$ such that $\cdot c \cap \cdot c' \text{ holds for all } c, c' \in C'$ and $\{ \langle h(c) \mid c \in C' \rangle \} = \cdot t$. In this case, we add edges $(c, e)$ for each $c \in C'$ (i.e. $\cdot c = C'$), we set $h(e) = t$, and for each $p \in t^\bullet$, we add to $C$ a condition $\langle e, p \rangle$, connected by an edge $(c, e)$.

Intuitively, a condition $\langle x, p \rangle$ represents the possibility of putting a token onto place $p$ through a particular firing sequence, while an event $\langle C', t \rangle$ represents a possibility of firing transition $t$ in a particular context.

Every firing sequence $\sigma$ is represented by a configuration of $U$; we denote this configuration $K(\sigma)$. Conversely, every configuration $E$ of $U$ represents one or several firing sequences ($K$ is not injective in general); these firing sequences are equivalent up to permutation of concurrent transitions. Their (common) resulting marking corresponds, due to the construction of $U$, to a reachable marking of $\mathcal{N}$. This marking is defined as $Mark(E) := \{ h(c) \mid c \in Cut(E) \}$.

2.2 Finite Complete Prefix

The unfolding $U$ of a finite safe Petri net $\mathcal{N}$ is infinite in general, but it shows some regularity because $\mathcal{N}$ has finitely many markings and two events $e$ and $e'$ having $Mark(\{e\}) = Mark(\{e'\})$ have isomorphic extensions.

It is known [16, 11] that one can construct a finite complete prefix $P$ of $U$, i.e. an occurrence net having a causally closed set $E'$ of events of $U$ which is sufficiently large for satisfying the following: for every reachable marking $M$ of $\mathcal{N}$ there exists a configuration $E$ of $P$ such that $Mark(E) = M$. One can even require that for each transition $t$ of $\mathcal{N}$ enabled in $M$, there is an event $\langle C, t \rangle \in E'$ enabled in $Cut(E)$.

The idea of the construction is to explore the future of only one among the events $e$ having equal $Mark(\{e\})$. The selected event is the one having minimal $\{e\}$ w.r.t. a so-called adequate order on the finite configurations of $U$. The others are flagged as cut-offs; they do not “contribute any new reachable markings”. These events are represented by dashed lines in Figure 1.

**Definition 5 (Adequate orders).** A strict partial order $\prec$ on the finite configurations of the unfolding of a safe Petri net $\mathcal{N}$ is called adequate if:

- it refines (strict) set inclusion $\subseteq$, i.e. $C \subseteq C'$ implies $C \prec C'$, and
- it is preserved by finite extensions, i.e. for every pair of configurations $C, C'$ such that $Mark(C) = Mark(C')$ and $C \prec C'$, and for every finite extension $D$ of $C$, the finite extension $D'$ of $C'$ which is isomorphic to $D$ satisfies $C \cup D \prec C' \cup D'$.

The initial definition of adequate orders [11] also requires that $\prec$ is well founded, but [6] showed that, for unfoldings of safe Petri nets, well-foundedness is a consequence of the other requirements.

Efficient tools [20, 14] exist for computing finite complete prefixes.
3 Goal-Oriented Model Reduction

The goal-driven unfolding relies on model reduction procedures which preserve minimal firing sequence to reach a given goal $g$. These reductions aim at removing as many transitions as possible among those that do not participate in any minimal firing sequence. This section details the properties required by our method and introduces several notations used in the rest of the paper.

Definition 6 (Minimal firing sequence). A firing sequence $t_1 \ldots t_n$ of a Petri net $\mathcal{N} = \langle P,T,F,M_0 \rangle$ visiting markings $M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} M_2 \ldots \xrightarrow{t_n} M_n$ is said to be cycling if it visits the same marking twice, i.e. $M_i = M_j$ for some $0 \leq i < j \leq n$. A minimal firing sequence of $\mathcal{N}$ to a goal $g$ is a firing sequence $t_1 \ldots t_n$ leading to $g$ for which there exists no different permutation$^1$ being a cycling firing sequence of $\mathcal{N}$.

For example with Petri net of Figure 1 and considering the goal $\{p_3,p_5\}$, $t_3t_0t'_1t_2t_3t'_2t'_3t_4$ is not minimal because its permutation $t_3t_0t'_1t_2t_3t'_2t'_3t_4$ is also feasible and visits the marking $\{p_3,p_0\}$ twice. Intuitively, the cycle $t'_2t_3$ can be removed. The minimal firing sequences of $\mathcal{N}$ to the goal are $t_3t_0t'_1t'_1, t_3t_0t'_2t'_2$ and their feasible permutations, for instance $t_3t_0t'_2t'_2t'_3$.

Remark. Alternatively, the goal can be seen not as a marking but simply as a set of places to be marked together, possibly with others. Then, one is looking for sequences reaching any marking $M$ with $g \subseteq M$. For minimality, we would then require additionally that no intermediate marking reached before the end of the sequence marks the places in $g$ (and the same for its permutations).

Definition 7 (Minimal configuration). A minimal configuration of a Petri net $\mathcal{N}$ to a goal $g$ is a configuration $E = K(\sigma)$ for some minimal firing sequence $\sigma$ of $\mathcal{N}$ to $g$. Notice that, since all the other $\sigma'$ such that $E = K(\sigma')$ are permutations of $\sigma$, they are all minimal.

Lemma 8. The goal $g$ is reachable iff it is reachable by a minimal firing sequence (and, consequently, by a minimal configuration).

Proof. Assume that $g$ is reachable by a non-minimal firing sequence $\sigma$. This means that $\sigma$ has a permutation $t_1 \ldots t_n$ which visits the same marking twice, i.e. $M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} M_2 \ldots \xrightarrow{t_n} M_n = g$ with $M_i = M_j$ and $i < j$. Then $g$ is also reachable by the strictly shorter sequence $t_1 \ldots t_j t_{j+1} \ldots t_n$. This operation can be iterated if needed; it always terminates and gives a minimal firing sequence which reaches the goal $g$.

Definition 9 (Reduction procedure, useless transitions). A reduction procedure useless-trs is a function which outputs, for a safe Petri net $\mathcal{N}$ and a goal $g \subseteq P$, a set useless-trs($\mathcal{N},g$) $\subseteq T$ of transitions of $\mathcal{N}$ which do not occur in any minimal firing sequence of $\mathcal{N}$ to goal $g$; for every minimal firing sequence $t_1 \ldots t_n$ to goal $g$, useless-trs($\mathcal{N},g$) $\cap \{t_1, \ldots, t_n\} = \emptyset$.

For example, let $\mathcal{N} = \langle P,T,F,M_0 \rangle$ be the Petri net of Figure 1. All the transitions occur in at least one minimal firing sequence to the goal $g = \{p_3,p_5\}$, so every reduction procedure outputs useless-trs($\mathcal{N},g$) = $\emptyset$. After firing $t_3t_0t'_3$, one reaches marking $\{p'_3,p'_5\}$ from which the only minimal firing sequences to $g$ are $t_1t'_1t'_2$ and $t_2t'_2t'_3$. Hence, a reduction

$^1$ Contrary to what is common in concurrency theory, we do not necessarily restrict to permutations of independent transitions w.r.t. an independence relation.
The principle of the goal-driven unfolding is that, for some events $e$, any firing sequence of 

$$\langle P, T, F, M_0 \rangle$$

may declare $t_0$, $t_3$ and $t'_3$ useless, or any subset of those.

Given a Petri net $\mathcal{N} = \langle P, T, F, M_0 \rangle$, $\mathcal{N} \setminus \text{useless-trs}(\mathcal{N}, g)$ denotes the reduced model $\langle P, T', F', M_0 \rangle$ where $T' = T \setminus \text{useless-trs}(\mathcal{N}, g)$ and $F' = F \cap ((P \times T') \cup (T' \times P))$.

Property 1. Every reduction procedure preserves reachability of the goal: $g$ is reachable in $\mathcal{N}$ iff it is reachable in $\mathcal{N} \setminus \text{useless-trs}(\mathcal{N}, g)$.

In the sequel, we aim at iterating the reduction procedures: starting from a model $\mathcal{N} = \langle P, T, F, M_0 \rangle$ and a goal $g$, we will apply the reduction to $\mathcal{N}$, then explore the reduced net $\mathcal{N} \setminus \text{useless-trs}(\mathcal{N}, g)$; later on, we will apply again the reduction from a reached state $M$ and compute $\text{useless-trs}(\mathcal{N}', g)$ with $\mathcal{N}' = \langle P, T, F, M \rangle \setminus \text{useless-trs}(\mathcal{N}, g)$ allowing to explore a further reduced net $\mathcal{N}' \setminus \text{useless-trs}(\mathcal{N}', g)$ from $M$. These iterated calls to the reduction procedure are justified by the following lemma.

Lemma 10. Any minimal sequence in $\mathcal{N} \setminus \text{useless-trs}(\mathcal{N}, g)$ is minimal in $\mathcal{N}$.

Proof. Any firing sequence of $\mathcal{N} \setminus \text{useless-trs}(\mathcal{N}, g)$ is a firing sequence of $\mathcal{N}$, and the minimality criterion does not depend on the set of transitions in $\mathcal{N}$. ▶

In the remainder of the paper, for a Petri net $\mathcal{N} = \langle P, T, F, M_0 \rangle$ and any set $I \subseteq T$ and marking $M$, we write $\text{useless-trs}(\mathcal{N}, g, M, I)$ for $\text{useless-trs}(\langle P, T, F, M \rangle \setminus I, g) \cup I$.

4 Goal-Driven Unfolding

In this section, we first show that model reduction can be performed during the unfolding of a safe Petri net $\mathcal{N}$ while preserving the minimal configurations to the goal. Next we present an algorithm to construct a finite goal-driven prefix which preserves the reachable markings of the goal-driven unfolding.

4.1 Guiding the Unfolding by a Model Reduction Procedure

The principle of the goal-driven unfolding is that, for some events $e$ in the unfolding (at discretion), a model reduction procedure $\text{useless-trs}$ is called and the transitions declared useless will not be considered in the future of $e$. More precisely, the reduction procedure is called on the marking $\text{Mark}([e])$ of the causal past of $e$.

Notice that the reduction procedure may already have been used on some events in the causal past of $e$. Then,

- even if $\text{useless-trs}$ is not called on $e$, information about useless transitions inherited from the causal predecessors of $e$ can be used (without calling the model reduction procedure), and this will already prune some branches in the future of $e$;
- if the reduction procedure is called on $\text{Mark}([e])$, it can take as input the model already reduced by the transitions declared useless after some event in the causal past of $e$.

Let $U = \langle C, E, G, C_0 \rangle$ be the full unfolding of a safe Petri net $\mathcal{N}$. Denote $E'$ the set of events on which the reduction procedure is called. The set $E'$ and the reduction procedure define the set of transitions $\text{Useless}(e)$ to be ignored in the future of an event $e \in E$. We define $\text{Useless}$ inductively as:

\[
\text{Useless}(e) = \begin{cases} 
\bigcup_{e' \in [e]} \text{Useless}(e') & \text{if } e \notin E' \\
\text{useless-trs} \left( \mathcal{N}, g, \text{Mark}([e]), \bigcup_{e' \in [e]} \text{Useless}(e') \right) & \text{if } e \in E'.
\end{cases}
\]
Thus, every event \( e = (C,t) \in E \) such that \( t \in \text{Useless}(e') \) for some \( e' \in [e] \), is discarded from the goal-driven unfolding. Denote \( E_{\text{Ignored}} \) the set of such events.

It remains to define the goal-driven unfolding as the maximal prefix of the full unfolding \( \mathcal{U} \) having no event in \( E_{\text{Ignored}} \). Since every discarded event automatically discards all its causal successors, the set of events remaining in the goal-driven unfolding \( \mathcal{U}_{\text{gd}} \) is

\[
E_{\text{gd}} \overset{\text{def}}{=} \{ e \in E \mid [e] \cap E_{\text{Ignored}} = \emptyset \}.
\]

Notice that the events and conditions of the goal-driven unfolding as defined above can be constructed inductively following the procedure described in Section 2, enriched so that it attaches the set \( \text{Useless}(e) \) to every new event \( e \).

\textbf{Theorem 11.} (proof in Appendix A\textsuperscript{2}) The goal-driven unfolding \( \mathcal{U}_{\text{gd}} \) preserves all minimal configurations from \( M_0 \) to the goal.

A direct corollary is that the goal is reachable in \( \mathcal{N} \) iff the goal-driven unfolding contains a configuration which reaches it.

Notice that the precise definition of minimal sequences/configurations is crucial here, and especially the fact that the reduction procedure preserves \textit{all} minimal sequences/configurations. Indeed, imagine a situation where the minimal firing sequences to the goal fire two concurrent transitions \( t_1 \) and \( t_2 \) and then one out of two possible transitions \( t_3 \) and \( t_4 \). A reduction procedure which would guarantee only the preservation of \textit{some} minimal firing sequence to the goal could declare \( t_3 \) useless when called after the event corresponding to \( t_1 \), and declare \( t_4 \) useless when called after \( t_2 \), thus preventing to reach the goal.

### 4.2 Goal-Driven Prefix

We now define a finite goal-driven prefix. Our Algorithm 1 relies on the theory of adequate orders [11] developed for unfoldings. Any adequate order on the configurations of the full unfolding can be used, but, since our goal-driven unfolding prunes some branches of the unfolding, we have to adapt the construction.

A prefix \( \mathcal{P} \) has the same structure as an unfolding, with an additional field \( \text{coff} \) for the set of cut-off events. As usual, the procedure PUTATIVE-GD-PREFIX extends iteratively the prefix \( \mathcal{P} = (C,E,G,C_0,\text{coff}) \). An extension is an event \( e = (C',t) \) with \( C' \subseteq C \) s.t. \( \forall c,c' \in C' \), \( c \text{ co } c' \), \( \{ h(c) \mid c \in C' \} = \#t \), and \( \forall(e',p) \in C' \), \( e' \notin \text{coff} \). Here the procedure maintains a map \( \Delta \) of transitions that can be ignored, and considers an extension \( e = (C',t) \) only if the transition \( t \) is not declared useless, i.e., \( t \) is absent from \( \Delta(c') \) for all pre-conditions \( c' \in C' \).

The difficult part is that, when an event \( e \) is declared cut-off because \( \text{Mark}([e]) = \text{Mark}([e']) \) for an event \( e' < e \), nothing guarantees that the transitions allowed after \( e \) are also allowed after \( e' \). Then, \( e \) and \( e' \) have the same future in the full unfolding, but not necessarily in the goal-driven unfolding.

Figure 2 illustrates this situation. Let the goal be \( g = \{ p_4, p_3 \} \). It can be reached by the firing sequences \( a(bb')^*c(bb')^*b \) or \( a'b(bb')^*c(bb')^*b \). Only those who do not take the cycle \( bb' \) are minimal, namely \( acb \) and \( a'b'cb \). Notice that all the transitions participate in at least one minimal firing sequence, so the model \( \mathcal{N} \) cannot be reduced from the initial marking (every

\textsuperscript{2} Appendices available at https://hal.archives-ouvertes.fr/hal-01392203/file/godunf.pdf
Algorithm 1 Algorithm for goal-driven prefix computation.

1: procedure Putative-GD-Prefix($\mathcal{N}, \Delta$) with $\mathcal{N} = \langle P, T, F, M_0 \rangle$
2: \quad $\mathcal{P} \leftarrow \langle C \leftarrow \{ \langle \bot, p \rangle \mid p \in M_0 \}, E \leftarrow \emptyset, G \leftarrow \emptyset, C_0 \leftarrow \{ \langle \bot, p \rangle \mid p \in M_0 \}, \text{coff} \leftarrow \emptyset \rangle$
3: repeat
4: \quad Let $e = \langle C', t \rangle$ be a $\ll$-minimal extension of $\mathcal{P}$ s.t. $t \notin \bigcup_{c \in C'} \Delta(c')$.
5: \quad $E \leftarrow E \cup \{ e \}$
6: \quad $C \leftarrow C \cup \{ \langle e, p \rangle \mid p \in t^* \}$
7: \quad $G \leftarrow G \cup \{ \langle c', e \rangle \mid c' \in C' \} \cup \{ \langle e, (c, p) \rangle \mid p \in t^* \}$
8: \quad if $\exists e' \in E$ s.t. $\text{Mark}([e]) = \text{Mark}([e'])$ then
9: \quad \quad coff $\leftarrow$ coff $\cup \{ e \}$ /$e$ is a cut-off event/
10: \quad end if
11: \quad for all $c \in \{ \langle e, p \rangle \mid p \in t^* \}$ s.t. $c \notin \Delta$ do
12: \quad \quad $\Delta(c) \leftarrow \text{Useless}(c, \Delta, \mathcal{P})$
13: \quad end for
14: until no extension exists
15: end procedure

16: procedure Post-$\Delta(\Delta, \mathcal{P})$ with $\mathcal{P} = \langle C, E, G, C_0, \text{coff} \rangle$
17: \quad $\Delta' \leftarrow \Delta$ /copy map $\Delta$/
18: \quad for $e \in E$ following $\ll$ order do
19: \quad \quad for all $c \in e^*$ do
20: \quad \quad \quad $\Delta'(c) \leftarrow \Delta'(c) \cap \text{Useless}(c, \Delta, \mathcal{P})$
21: \quad \quad end for
22: \quad \quad if $\exists e' \in E \setminus \text{coff}$ s.t. $\text{Mark}([e]) = \text{Mark}([e'])$ then
23: \quad \quad \quad for all $c' \in \text{Cut}([e'])$ with $c \in \text{Cut}([e])$ and $h(c) = h(c')$ do
24: \quad \quad \quad \quad $\Delta'(c') \leftarrow \Delta'(c') \cap \Delta'(c)$
25: \quad \quad \quad end for
26: \quad \quad end if
27: \quad end for
28: end procedure

29: procedure GD-Prefix($\mathcal{N}$) with $\mathcal{N} = \langle P, T, F, M_0 \rangle$
30: \quad $\Delta' \leftarrow \{ \langle \bot, p \rangle \mid p \in M_0 \}$
31: repeat
32: \quad $\Delta \leftarrow \Delta'$ /copy map $\Delta'/\$
33: \quad $\Delta, \mathcal{P} \leftarrow$ Putative-GD-Prefix($\mathcal{N}, \Delta$) /can add new entries in $\Delta$/
34: \quad $\Delta' \leftarrow$ Post-$\Delta(\Delta, \mathcal{P})$
35: until $\Delta' = \Delta$
36: end procedure
The model reduction procedure will output `useless-trs(\mathcal{N}, g) = \emptyset`). On the other hand, if transition $a$ is fired, we reach marking $\{p_1, p_2\}$ from which $b', a$ and $a'$ become useless.

Now, observe the branching process on the right of Figure 2 (it is a prefix of the unfolding $\mathcal{U}$ of $\mathcal{N}$). Notice that the causal past $[e]$ of the event $e$ labeled $a'$ and the causal past of the event $e'$ labeled $b$ reach the same marking $\text{Mark}([e]) = \text{Mark}([e']) = \{p_1, p_3\}$. Moreover, an adequate order on the configurations of $\mathcal{U}$ may order them as $[e'] < [e]$. Consequently, $e$ is a cut-off and the minimal configuration $K(a'b'cb)$ is not represented in the finite prefix.

Following the idea of the proof of completeness of finite prefixes based on adequate orders, we can indeed shift the extension $b'cb$ of $[e]$ (in gray on the right of Figure 2) to the isomorphic extension of $[e']$ (also in gray on the figure). We get the configuration $K(abb'cb)$, which reaches the goal as well. But this configuration is not minimal any more because it executes the cycle $bb'$; the marking reached after $a$ is the same as the marking reached after $abb'$. Actually, the model reduction procedure called from the event labeled $a$ may very well have declared $b'$ useless. Consequently, $K(abb'cb)$ would not be represented in the prefix. We correct this by allowing after $[e']$ all the transitions that were allowed after $[e]$.

The difficulty in the definition and in the computation of a finite prefix $\mathcal{P}_{gd}$ of $\mathcal{U}$ which preserves the markings reachable in $\mathcal{U}_{gd}$ is to allow in the future of an event $e'$ all the transitions that are useful for at least one of all the configurations which are shifted to $[e']$ by the mechanics described above. The first answer to this problem is to allow after $[e']$ all the transitions that were allowed after $[e]$. This solves the problem of an event consuming only post-conditions of $e'$, like the occurrence of $b'$ after $e$ in our example of Figure 2: its corresponding event after $e'$ is now allowed. However, this is not sufficient in general: an event $f$ consuming a post-condition of $e$ may also consume other conditions which are created by events concurrent to $[e]$. Such event $f$ has a corresponding $f'$ in the future of $e'$, consuming conditions which are available after firing a configuration of the form $[e'] \cup \mathcal{E}'$ for some $\mathcal{E}'$ concurrent to $[e']$. We need the transition $t = h(e) = h(e')$ to be allowed after all the conditions consumed by $f'$. In the case of a condition $e' \in \mathcal{E}' \setminus \text{Cut}([e'])$, our procedure ensures this as follows: if it calls the model reduction procedure after the event...
where $E$ version of

\begin{itemize}
\item $\mathcal{E}$, it also calls it on the marking $Mark([e'] \cup [e'])$ which equals $Mark([e] \cup [e])$. Hence, if $t$ is needed after $[e] \cup [e]$, it will also be allowed after $e'$.
\end{itemize}

Therefore, when applying the reduction procedure after a configuration $\mathcal{E}$, we also take into account a set $Alt(\mathcal{E})$ of alternating configurations defined inductively as:

- $\mathcal{E} \in Alt(\mathcal{E})$
- $\forall \mathcal{E}' \in Alt(\mathcal{E}), \forall e, e' \in E$ such that $[e] \triangleright [e']$ and $Mark([e]) = Mark([e'])$, if $Cut([e']) \cap \mathcal{E}^* \neq \emptyset$ and $[e'] \cup \mathcal{E}'$ is conflict free, then $[e'] \cup \mathcal{E}' \in Alt(\mathcal{E})$.

However, in practice, during the computation of the goal-driven prefix, $Alt(\mathcal{E})$ will be computed on the events and configurations derived so far, hence ignoring events later added in the prefix. Also, as explained above, when an event $e$ is stated cut-off because of a $\prec$-smaller event $e'$, we allow after $e'$ all the transitions allowed after $e$; but this implies reconsidering some new extensions of $e'$.

For these reasons, the procedure GD-Prefix($N$) presented in Algorithm 1 iterates the computation of a putative prefix, progressively refining an over-approximation of transitions to ignore (map $\Delta$), by identifying a posteriori the transitions that should not have been ignored.

At each iteration, the procedure PUTATIVE-GD-PREFIX($N, \Delta$) computes a putative prefix, relying on the previous value of the map $\Delta$ of transitions that can be ignored. Essentially, the prefix $P$ obtained at the first iteration is the naive prefix of $U_{gd}$ (prefix without the gray parts on the example of Figure 2).

Once a putative prefix has been computed, we verify a posteriori if its related map $\Delta$ is correct. This is done by re-computing $\Delta$ using the procedure POST-$\Delta(\Delta, P)$, this time taking into account all the events in $P$ (line 34). By construction, the resulting $\Delta'$ can only allow more transitions than $\Delta$. If $\Delta'$ differs from $\Delta$, a new putative prefix is computed according to the corrected $\Delta'$.

The procedure POST-$\Delta(\Delta, P)$ takes the $Alt([e])$ into account by the way of a modified version of Useless($e$), now defined on conditions rather than on events. Given a condition $c \in \mathcal{E}^*$ in a prefix $P$,

\begin{equation}
Useless(c, \Delta, P) = \begin{cases}
\bigcup_{c' \in \mathcal{E}^*} \Delta(c') & \text{if } e \notin E' \\
\bigcap_{c' \in Alt([e])} \text{useless-trs} (N, g, Mark(C^*), \bigcup_{c' \in \mathcal{E}^*} \Delta(c')) & \text{if } e \in E',
\end{cases}
\end{equation}

where $E'$ is the set of events triggering an explicit reduction (Section 4.1).

This iterative construction necessarily terminates (Lemma 12, proof in Appendix B2) and converges to a unique finite prefix $P_{gd}$. Regarding complexity, putting aside the call to model reduction, whereas all the structures are finite, $Alt(\mathcal{E})$ can have an exponential numbers of configurations due to multiple combinations of configurations sharing an intersection.

**Lemma 12.** The procedure GD-Prefix($N$) terminates.

Notice that $P_{gd}$ may contain events that are not in $E_{gd}$. Hence, goal-driven prefix is a prefix of $U$, but not necessarily a prefix of $U_{gd}$. This is the case of the event labeled $b'$ after $e'$, as we discussed above for the example in Figure 2.

Theorem 13 (proof in Appendix C2) states completeness of $P_{gd}$ w.r.t. minimal configurations. Thus, the goal-driven prefix preserves the reachability of the goal. One can finally remark that, by construction, $P_{gd}$ contains at most one non-cutoff event per reachable marking, assuming the adequate order $\prec$ is total.

**Theorem 13.** For every configuration $\mathcal{E}$ of $U_{gd}$ and for every single-event extension $\{f\}$ of $\mathcal{E}$ such that $\mathcal{E} \cup \{f\}$ is a prefix of a minimal configuration to the goal, there exists
a configuration $\mathcal{E}'$ in the goal-driven prefix and a single-event extension $\{f'\}$ of $\mathcal{E}'$ with $\text{Mark}(\mathcal{E}) = \text{Mark}(\mathcal{E}')$ and $h(f) = h(f')$.

Example

Let us consider the Petri net of Figure 2(left) with the goal $\{p_4, p_3\}$.

The goal-driven unfolding can lead to the branching process of Figure 2(right) where the striped transition $b'$ has been removed. Indeed, after transition $a$, transition $b'$ is declared useless as it is not part of any minimal configuration extending $\mathcal{K}(a)$. Therefore 3 maximal configurations are remaining in the goal-driven unfolding: the two minimal configurations $\mathcal{K}(acb)$ and $\mathcal{K}(a'b'c'b)$, and the configuration $\mathcal{K}(ab)$ which does not reach the goal.

The goal-driven prefix can lead to the branching process of Figure 2(right) where the event $e$ is cut-off (because of $e'$), and therefore its future events are ignored, and where one of the two remaining events firing transition $c$ is declared cut-off (because of the other one). Although the $b'$ transition can be declared useless after $\mathcal{K}(a)$ (and hence $\mathcal{K}(ab)$), the cut-off of $e$ will remove $b'$ from the set of ignored transitions of the conditions matching with $p_1$ and $p_3$ on the cut of $\mathcal{K}(ab)$. Therefore, the events and conditions in the left gray area will be added to the prefix, from which all the minimal configurations can be identified.

5 Experiments

In this section, we instantiate our goal-driven unfolding with the goal-oriented model reduction introduced in [17] for automata networks. We compare the size of the complete prefix with the goal-driven prefix on different Petri net models of biological signalling and gene regulatory networks. In general, such networks gather dozens to thousands nodes having sparse interactions (each node is directly influenced by a few other nodes), which call for concurrency-aware approaches to cope with the state space explosion. We took the networks from systems biology literature, specified as Boolean or automata networks: each node is modelled by an automaton, where states model its activity level, most often being binary (active or inactive). The Petri nets are encodings of these automata networks which ensure bisimilarity [5].

5.1 Implementation

In practice, instead of computing putative prefixes from scratch as it is described in Algorithm 1, our implementation for the goal-driven prefix\(^3\) iteratively corrects the putative prefix by propagating transitions missed in the previous iteration. At this stage, it does not use any particular optimization [1], our primary objective being to compare the size of the resulting prefixes. In order to obtain a proper comparison [15], our implementation uses the same arbitrarily-fixed ordering for the complete and goal-driven prefixes extensions.

The computation of $\text{useless-trs}(N, g, M, I)$ relies on the goal-oriented reduction of asynchronous automata networks introduced in [17]. This method is based on a static analysis of causal dependencies of transitions and an abstract interpretation of traces which allow to collect all the transitions involved in the minimal configurations to the goal: non-collected transitions can then be ignored. The complexity of the reduction is polynomial with the number of automata and transitions, and exponential with the number of states in individual

\(^3\) Code and models available at http://loicpauleve.name/godunf.tbz2
Table 1: Benchmarks of the goal-driven w.r.t. complete prefix of 1-safe Petri nets. For each model, the number of places $|P|$ and transitions $|T|$ is given. The strategy decides when the model reduction should be performed; the number of calls to the reduction procedure is indicated in the column “Nb reductions”. Computation times were obtained on an Intel® Core™ i7 3.4GHz CPU with 16GB RAM. N/A: Non Applicable; ∗: out-of-memory computation (with mole [20], with the same ordering for extensions as our implementation), the indicated prefix size is only a lower bound.

<table>
<thead>
<tr>
<th>Model</th>
<th>Prefix</th>
<th>Strategy</th>
<th>Prefix size</th>
<th>Time</th>
<th>Nb reductions</th>
</tr>
</thead>
<tbody>
<tr>
<td>RB/E2F</td>
<td>complete</td>
<td>N/A</td>
<td>15,210</td>
<td>24s</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>goal-driven</td>
<td>always</td>
<td>112</td>
<td>0.5s</td>
<td>136</td>
</tr>
<tr>
<td>T-LGL</td>
<td>complete</td>
<td>N/A</td>
<td>$\geq 1,900,000^*$</td>
<td>OT^*</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>goal-driven</td>
<td>always</td>
<td>17</td>
<td>0.3s</td>
<td>17</td>
</tr>
<tr>
<td>VPC</td>
<td>complete</td>
<td>N/A</td>
<td>44,500</td>
<td>176s</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>goal-driven</td>
<td>always</td>
<td>1,827</td>
<td>2h</td>
<td>16,009</td>
</tr>
<tr>
<td></td>
<td></td>
<td>first 1,000</td>
<td>2,036</td>
<td>60s</td>
<td>1,000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>level ≤ 2</td>
<td>2,400</td>
<td>7s</td>
<td>38</td>
</tr>
</tbody>
</table>

automata (i.e., number of qualitative states of nodes). As shown in [17], the method can lead to drastic model reductions and can be executed in a few hundredths of a second on networks with several hundreds of nodes. Appendix D gives a brief summary of the main principles of the goal-oriented model reduction of [17].

Compared to the results obtained in [17], we expect to obtain much stronger reduction of the dynamics as our goal-driven unfolding applies the reduction “on-the-fly” instead of only at the initial state. To that aim, we will compare the size of the complete prefix, which relies only on [17], with the size of the goal-driven prefix, introduced in this paper.

We applied the goal-driven unfolding to 1-safe Petri net encodings of the automata networks, where there is one place for each local state of each automaton, and a one-to-one relationship between transitions. The places corresponding to states of a same automaton are mutually exclusive by construction. Future work may consider goal-driven unfolding of products of transition systems [10].

The goal-driven prefix we define in this paper supports calling the model reduction procedure at discretion: even if it has a low computational cost, performing the model reduction after each event may turn out to be very time consuming. Our prototype implements simple strategies to decide when the call to the model reduction should be performed: after each event; only for the first $n$ events; and only for events up to a given level in the unfolding.

5.2 Benchmarks

Given a Petri net with an initial marking $M_0$ and a goal $g$, we first compute the goal-oriented model reduction from initial marking (useless-trs ($\mathcal{N}, g, M_0, \emptyset$)). The resulting net is then given as input to the unfolding, either with the complete finite prefix computation, or with the goal-driven. Therefore, the difference in the size of the prefixes obtained is due only to transition exclusions after at least one event.

Table 1 summarizes the benchmarks between complete and goal-driven prefix on different models of biological networks. The size of a prefix is the number of its non-cutoff events. “RB/E2F” is a model of the cell cycle [4]; “T-LGL” is a model of survival signaling in large granular lymphocyte leukemia [23]; and “VPC” is a model for the specification of vulval precursor cells and cell fusion control in Caenorhabditis elegans [22]. For each model, the initial marking and goal correspond to biological states of interest (checkpoints or
differentiated states). All these models have different network topology and dynamical features, but all include loops, and in general, correspond to automata networks where automata have few internal states, and each transition is conditioned by a few automata compared to the network size.

On these models, the goal-driven prefix shows a significant size reduction, while containing all the minimal configurations. The number of reductions can be larger than the size of the prefix as it accounts for the intermediate putative prefixes (as explained in Section 4.2). For the “VPC” model, we applied several strategies for deciding when the model reduction should be called. In this case, the systematic model reduction led to some re-ordering of the extensions and cut-offs declaration, which required numerous additional calls to the model reduction procedure. This motivates the design of heuristics to estimate when a model reduction should be performed. For the “T-LGL” model, it was impossible to compute the complete finite prefix using only the reduction of [17] on the initial state, whereas the goal-driven cuts most of the configurations and produces a very concise prefix. Such a behaviour can be explained by large transient cycles prior to the goal reachability, which are avoided by the use of model reduction during the prefix computation.

6 Conclusion

We introduced the goal-driven unfolding of safe Petri nets for identifying efficiently all the minimal configurations that lead to a given goal. The goal can be a marking of the net, or any partially specified marking, and notably a single marked place. The goal-driven unfolding relies on an external reduction method which identifies transitions that are not part of minimal configuration for the goal reachability. Such useless transitions are then skipped by the unfolding. The computation of a goal-driven prefix requires a particular treatment of cut-offs to ensure that all the markings reachable in the goal-driven unfolding are preserved. The resulting goal-driven prefix contains fewer events prefix than reachable markings, due to the total adequate order, as well as for classical finite complete prefix.

We instantiated our approach with a goal-oriented reduction method for automata networks introduced in [17]. Our experiments on different models of biological systems show a significant reduction of the prefix when driven by the goal. In our framework, the reduction procedure can be applied at discretion, and many possible heuristics could be embedded to decide when the reduction is timely, which impacts both the execution time and the size of the prefix.

Because our method considers the model reduction procedure as a blackbox, it can directly take advantage of any improvements of model reduction procedures which preserve minimal configurations.

Future work will explore the combination with the semi-adequate ordering of configurations of directed unfolding [3] as it may reduce the need for propagating transitions allowed by a cut-off event. Finally, we are considering implementing the goal-driven unfolding within Mole [20].

References


Probabilistic Automata of Bounded Ambiguity

Nathanaël Fijalkow¹, Cristian Riveros², and James Worrell³

¹ The Alan Turing Institute of Data Science, London, UK and University of Warwick, Coventry, UK
nfijalkow@turing.ac.uk
² Pontificia Universidad Católica de Chile, Santiago, Chile cristian.riveros@uc.cl
³ University of Oxford, Oxford, UK james.worrell@cs.ox.ac.uk

Abstract

Probabilistic automata are a computational model introduced by Michael Rabin, extending non-deterministic finite automata with probabilistic transitions. Despite its simplicity, this model is very expressive and many of the associated algorithmic questions are undecidable. In this work we focus on the emptiness problem, which asks whether a given probabilistic automaton accepts some word with probability higher than a given threshold. We consider a natural and well-studied structural restriction on automata, namely the degree of ambiguity, which is defined as the maximum number of accepting runs over all words. We observe that undecidability of the emptiness problem requires infinite ambiguity and so we focus on the case of finitely ambiguous probabilistic automata.

Our main results are to construct efficient algorithms for analysing finitely ambiguous probabilistic automata through a reduction to a multi-objective optimisation problem, called the stochastic path problem. We obtain a polynomial time algorithm for approximating the value of finitely ambiguous probabilistic automata and a quasi-polynomial time algorithm for the emptiness problem for ²-ambiguous probabilistic automata.

1998 ACM Subject Classification F.1.1 Models of Computation

Keywords and phrases Probabilistic Automata, Emptiness Problem, Stochastic Path Problem, Multi-Objective Optimisation Problems

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2017.19

1 Introduction

Probabilistic automata are a simple and natural extension of non-deterministic automata that were introduced by Rabin [16]. Syntactically, a probabilistic automaton is a non-deterministic finite automaton in which each edge is annotated by a probability. Such an automaton associates to every word a value between 0 and 1, which is the total probability that a run on the word ends in an accepting state. We call this the acceptance probability of the word.

Despite their simplicity, probabilistic automata are very expressive and have been widely studied. Unfortunately the price of this expressiveness is that almost all natural decision problems are undecidable. Consequently, various approaches based on restricting resources such as structure, dimension, or randomness have been studied [5, 8, 7, 4].
In this paper, we look at probabilistic automata of bounded ambiguity, where the ambiguity of a word is the number of accepting runs. We say that a probabilistic automaton is $f$-ambiguous, for a function $f : \mathbb{N} \to \mathbb{N}$, if every word of length $n$ has at most $f(n)$ accepting runs. (Note that ambiguity is a property of the underlying nondeterministic finite automata, and is independent of the transition probabilities.) This restriction has been extensively studied in automata theory; in particular, the landmark paper of Weber and Seidl [18] gives respective structural characterisations of the classes finitely, polynomially, and exponentially ambiguous nondeterministic finite automata, from which polynomial-time algorithms are obtained for deciding membership in each of these classes.

We focus on the most natural and well-studied problem for probabilistic automata, called the emptiness problem: given a probabilistic automaton and a threshold, does there exist a word accepted with probability at least a given threshold? Using a classical construction, we first observe that the emptiness problem is already undecidable for polynomially ambiguous probabilistic automata. We are thus led to focus on finitely ambiguous probabilistic automata.

We study the complexity of the emptiness problem on various classes of finitely ambiguous probabilistic automata. For each positive integer $k$ we consider the class of $k$-ambiguous probabilistic automata, i.e., automata with at most $k$ accepting runs on any word. More generally we fix a polynomial $p$ and consider the class of automata whose ambiguity is at most $p(m)$, where $m$ is the number of states. More generally still, bearing in mind that the ambiguity can be exponential in the number of states, we have the class of all finitely ambiguous automata.

Our main results are as follows. We show that the emptiness problem for finitely ambiguous probabilistic automaton is, respectively:

- in \textsc{NEXPTIME} and \textsc{PSPACE}-hard for the class of all finitely ambiguous automata;
- \textsc{PSPACE}-complete for the class of probabilistic automata with ambiguity bounded by a fixed non-constant polynomial in the number of states.
- in \textsc{NP} for the class of $k$-ambiguous probabilistic automata, for every positive integer $k$.
- in quasi-polynomial time for the class of $2$-ambiguous probabilistic automata.

Naturally associated with the emptiness problem we have the function problem of computing the value of a probabilistic automaton, that is, the supremum over all words of the acceptance probability of a word. Here we show:

- for the class of all finitely ambiguous probabilistic automata, there is no polynomial-time approximation algorithm for the value problem unless $\textsc{P} = \textsc{NP}$.
- for the class of $k$-ambiguous probabilistic automata, the value is approximable up to any multiplicative constant in polynomial time.

The starting point to prove these results is to give an upper bound on the length of a shortest word whose probability exceeds a given threshold. More precisely, we show that for a $k$-ambiguous probabilistic automaton with $n$ states there is a maximum-probability word of length at most $n^k$. More generally, we show that for a finitely ambiguous probabilistic automaton with $n$ states, there is a maximum-probability word of length at most $n!$. The latter result easily leads to a \textsc{PSPACE} upper bound for the emptiness problem in the case the ambiguity is bounded by a fixed polynomial in the number of states. Most of the remainder of the paper is devoted to the case of $k$-ambiguous automata for a fixed $k$.

We give a polynomial-time reduction from the emptiness problem for $k$-ambiguous probabilistic automata to a multi-objective optimisation problem, which we call the $k$-stochastic path problem. Using this reduction, we obtain a polynomial-time algorithm for approximating the value of a $k$-ambiguous probabilistic automata, and a quasi-polynomial time algorithm for the emptiness problem of $2$-ambiguous probabilistic automata.
2 Preliminaries

Let $\Sigma$ be a finite alphabet. For any word $w \in \Sigma^*$, we denote by $|w|$ its length. A distribution is a function $\delta : Q \to [0,1]$ such that $\sum_{q \in Q} \delta(q) = 1$. The set of distributions over $Q$ is denoted $\mathcal{D}(Q)$.

Probabilistic automata. A probabilistic automaton is a tuple $\mathcal{P} = (Q, q_{in}, \Delta, F)$, where $Q$ is a finite set of states, $q_{in}$ is the initial state, $\Delta : Q \times A \to \mathcal{D}(Q)$ is the transition function, and $F$ is the set of accepting states. Given a word $w = a_1 \cdots a_n$, a run $\rho$ over $w$ is a sequence of states $q_0, q_1, \ldots, q_n$. The probability of such a run is $P(\rho) = \prod_{i \in \{1,\ldots,n\}} \Delta(q_{i-1}, a_i)(q_i)$.

A run $\rho$ is accepting if it starts in $q_{in}$, satisfies $P(\rho) > 0$, and finishes in an accepting state, i.e. a state in $F$. We denote by $\text{Run}_\mathcal{P}(p \xrightarrow{w} q)$ the set of runs $\rho$ over $w$ starting in $p$ and finishing in $q$ with $P(\rho) > 0$. The number $P(p \xrightarrow{w} q)$ is the probability to go from $p$ to $q$ reading $w$, defined as the sum of the probabilities of its runs, namely:

$$P(p \xrightarrow{w} q) = \sum_{\rho \in \text{Run}_\mathcal{P}(p \xrightarrow{w} q)} P(\rho).$$

A run $\rho$ is accepting if it starts in $q_{in}$, satisfies $P(\rho) > 0$, and finishes in an accepting state, i.e. a state in $F$. We denote by $\text{Run}_\mathcal{P}(w)$ the set of accepting runs over $w$. The probability of $w$ over $\mathcal{P}$ is defined as the sum of the probabilities of its accepting runs by:

$$P(w) = \sum_{\rho \in \text{Run}_\mathcal{P}(w)} P(\rho).$$

Ambiguity. In this paper, we consider different subclasses of probabilistic automata, obtained by restrictions on ambiguity. More specifically, we say that:

- $\mathcal{P}$ is unambiguous if every word $w$ has at most one accepting run, i.e. $|\text{Run}_\mathcal{P}(w)| \leq 1$.
- $\mathcal{P}$ is $k$-ambiguous if every word $w$ has at most $k$ accepting runs, i.e. $|\text{Run}_\mathcal{P}(w)| \leq k$.
- $\mathcal{P}$ is finitely ambiguous, if there exists $k$ such that $\mathcal{P}$ is $k$-ambiguous.
- $\mathcal{P}$ is polynomially ambiguous, if there exists a polynomial $P$ such that for every word $w$, we have $|\text{Run}_\mathcal{P}(w)| \leq P(|w|)$.

Depending on whether the polynomial $P$ is linear or quadratic, we say that a polynomial ambiguous automaton $\mathcal{P}$ is linearly ambiguous or quadratically ambiguous, respectively. Note that it is decidable in polynomial time whether a probabilistic automaton $\mathcal{P}$ is unambiguous, finitely ambiguous, or polynomially ambiguous [18]. Furthermore, a consequence of the results of [18] is that an automaton which is not finitely ambiguous is at least linearly ambiguous.

Emptiness problem and value. Let $\mathcal{P}$ be a probabilistic automaton and $c$ a threshold. Following Rabin [16], we define the threshold language induced by $\mathcal{P}$ and $c$ as:

$$L^{>c}(\mathcal{P}) = \{ w \in \Sigma^* \mid P(w) > c \}.$$  

The emptiness problem asks, given a probabilistic automaton $\mathcal{P}$ and a threshold $c$, whether the language $L^{>c}(\mathcal{P})$ is non-empty, that is, whether there exists a word $w$ such that $P(w) > c$.

A related function problem is to compute the value of a probabilistic automaton $\mathcal{P}$, defined by $\text{val}(\mathcal{P}) = \sup_{w \in \Sigma^*} P(w)$. Note that the emptiness problem is equivalent to asking whether $\text{val}(\mathcal{P}) > c$. 

CONCUR 2017
Probabilistic Automata of Bounded Ambiguity

Figure 1 This probabilistic automaton computes \( \text{bin} \).

3 Undecidability for Polynomially Ambiguous Probabilistic Automata

In this section, we show undecidability results for polynomially ambiguous probabilistic automata, which justifies the focus of our paper on finitely ambiguous probabilistic automata.

\[ \text{Theorem 1.} \quad \text{The emptiness problem is undecidable for quadratically ambiguous probabilistic automata.} \]

Undecidability of the emptiness problem has long been known for general probabilistic automata [15, 2, 9]. However, the automata involved in the proof have exponential ambiguity. We show that the ideas can be reused to obtain Theorem 1. The key ingredient in the undecidability proof of Bertoni [2] (see [9] for a simple exposition of the ideas) is the construction of a probabilistic automaton computing the value of a rational number given in binary with least significant digit on the left:

\[
\text{bin}^R(a_1 \cdots a_n) = \sum_{i=1}^{n} \frac{a_i}{2^{n-i+1}}.
\]

The automaton proposed by Bertoni has exponential ambiguity. However, it is possible to construct a linearly ambiguous probabilistic automaton computing the same function but reversing the input:

\[
\text{bin}(a_1 \cdots a_n) = \sum_{i=1}^{n} \frac{a_i}{2^i}.
\]

The automaton is represented in Figure 1. Once this automaton is known, it is easy to prove the undecidability of the emptiness problem following the proof scheme of Bertoni, streamlined by Gimbert and Oualhadj [9]:

First, show that the following problem is undecidable: given a linearly ambiguous probabilistic automaton \( \mathcal{P} \), does there exist a word \( w \) such that \( \mathcal{P}(w) = \frac{1}{2^k} \)? The proof is by reduction from Post’s Correspondence Problem (PCP), which can be defined as follows: given a pair of monoid homomorphisms \( \varphi_1, \varphi_2 : \Sigma^* \to \{0,1\}^* \), does there exist a word \( w \) such that \( \varphi_1(w) = \varphi_2(w) \)? Using the automaton above computing bin, it is easy to construct a probabilistic automaton \( \mathcal{P} \) of linear ambiguity such that \( \mathcal{P}(w) - \frac{1}{2} = \frac{1}{2} (\text{bin}(\varphi_1(w)) - \text{bin}(\varphi_2(w))) \). Since the function bin is (essentially\(^1\)) injective, \( \mathcal{P}(w) = \frac{1}{2} \) is equivalent to \( \varphi_1(w) = \varphi_2(w) \), proving the correctness of the reduction.

\(^1\) One needs to ensure that the last letter is a 1, which is achieved using a small modification of \( \varphi_1 \) and \( \varphi_2 \)
Second, show that the emptiness problem is undecidable by reduction to the problem above. Given a linearly ambiguous probabilistic automaton $\mathcal{P}$, one can construct a quadratically ambiguous probabilistic automaton $\mathcal{P}'$ such that $\mathcal{P}'(w) = \mathcal{P}(w \cdot (1 - \mathcal{P}(w)))$. Since for $x$ in $[0, 1]$, the following equivalence holds: $x = \frac{1}{2}$ if, and only if, $x \cdot (1 - x) \geq \frac{1}{2}$, the first undecidability result implies the undecidability of the emptiness problem for quadratically ambiguous probabilistic automata.

Given a probabilistic automaton $\mathcal{P}$, we say that a threshold $c$ is isolated if there exists $\varepsilon > 0$ such that for all words $w$, we have $|\mathcal{P}(w) - c| > \varepsilon$. Rabin [16], proved that if a threshold $c$ is isolated then the corresponding language $L^{>\varepsilon}(\mathcal{P})$ is regular. The isolation problem asks to determine whether a given threshold is isolated for a given automaton. This problem was shown to be undecidable by Bertoni [2]. We can refine the result of [2] to obtain:

\textbf{Theorem 2.} \textit{The isolation problem is undecidable for linearly ambiguous probabilistic automata.}

\textbf{Proof.} We construct a reduction from a variant of the Post Correspondence’s Problem, called the infinite PCP, and shown to be undecidable in [17]. The problem asks, given two homomorphisms $\varphi_1, \varphi_2 : \Sigma^* \rightarrow \{0, 1\}^*$, does there exist an infinite word $w$ in $\Sigma^\omega$ such that $\varphi_1(w) = \varphi_2(w)$ (where $\varphi_1, \varphi_2$ are extended to continuous $\Sigma^\omega$ maps on with respect to product topology). We first observe that equivalently, we ask whether for every $\varepsilon > 0$ there exists $w$ in $\Sigma^*$ such that $|\text{bin}(\varphi_1(w)) - \text{bin}(\varphi_2(w))| \leq \varepsilon$.

Indeed, if there exists an infinite word $w$ such that $\varphi_1(w) = \varphi_2(w)$, then the sequences obtained by considering the images under $\varphi_1$ and $\varphi_2$ of prefixes of $w$ have arbitrarily long common prefixes, so the difference of their binary values converges to 0. Conversely, assume that for any $\varepsilon > 0$ there exists a finite word $w$ such that $|\text{bin}(\varphi_1(w)) - \text{bin}(\varphi_2(w))| \leq \varepsilon$, then we construct a solution to the infinite PCP using König’s lemma. To this end, for each $n$ let $w_n$ be a finite word such that $|\text{bin}(\varphi_1(w_n)) - \text{bin}(\varphi_2(w_n))| < 2^n$, i.e., such that $\varphi_1(w_n)$ and $\varphi_2(w_n)$ coincide on the first $n$ letters. Applying König’s Lemma to the infinite tree defined by the prefix closure of the set $\{w_n \mid n \geq 0\}$ (i.e., each node in the tree is the prefix of some word $w_n$), there exists an infinite word $w$ such that $\varphi_1(w) = \varphi_2(w)$.

We now construct the reduction from the infinite PCP to the isolation problem for linearly ambiguous probabilistic automata. Given two homomorphisms $\varphi_1$ and $\varphi_2$ we construct the linearly ambiguous probabilistic automaton $\mathcal{P}$ such that for every $w$ in $\Sigma^*$,

$$\mathcal{P}(w) = \frac{1}{2} \left( \text{bin}(\varphi_1(w)) + 1 - \text{bin}(\varphi_2(w)) \right).$$

Then\footnote{The same trick as for the proof above is required, to ensure that bin is injective.} for every $\varepsilon > 0$ there exists $w$ in $\Sigma^*$ such that $|\text{bin}(\varphi_1(w)) - \text{bin}(\varphi_2(w))| \leq \varepsilon$ if, and only if, $|\mathcal{P}(w) - \frac{1}{2}| \leq \varepsilon$.

An automaton is either finitely ambiguous, or at least linearly ambiguous. We proved undecidability results for linearly and quadratically ambiguous automata; the focus of the present paper is on decidability results for finitely ambiguous automata.

\section{Decidability and Complexity of Finitely Ambiguous Probabilistic Automata}

In this section we study threshold languages and the emptiness problem for finitely ambiguous probabilistic automata. We start by showing regularity of the threshold language $L^{>\varepsilon}(\mathcal{P})$ for
Probabilistic Automata of Bounded Ambiguity

a finitely ambiguous probabilistic automaton $P$ and threshold $c$. A classical result, due to Rabin [16], shows that the threshold languages need not be regular in general. Unfortunately the proof of regularity, while constructive, is not useful for determining the complexity of the emptiness problem. However we are able to give a direct simple argument that bounds the length of witnesses for the emptiness problem. We then use these bounds to analyse the complexity of the emptiness problem.

**Theorem 3.** Let $P$ be a finitely ambiguous probabilistic automaton and $c$ a threshold. Then $L_{\leq c}(P)$ is a regular language.

**Proof.** Consider the set $\mathbb{N}^k$ under the pointwise order. Recall that $I \subseteq \mathbb{N}^k$ is an ideal if it is downward closed and directed. Every ideal $I$ has the form

$$I = \{(n_1, \ldots, n_k) \in \mathbb{N}^k : n_{i_1} \leq a_1 \land \ldots \land n_{i_s} \leq a_s\}$$

(1)

for certain indices $1 \leq i_1 < \ldots < i_s \leq k$ and natural numbers $a_1, \ldots, a_s$. From the fact that $\mathbb{N}^k$ is a well-quasi-order it follows that every downward closed subset of $D \subseteq \mathbb{N}^k$ can be written as a finite union of ideals. Indeed such a decomposition can easily be computed from the finite set of minimal elements of $\mathbb{N}^k \setminus D$ [12].

Let $P = (Q, q_0, \Delta, F)$ be a finitely ambiguous probabilistic automaton with transition function $\Delta : Q \times \Sigma \to D(Q)$. Say that a triple $(p, a, q) \in Q \times \Sigma \times Q$ is an edge if $\Delta(p, q)(a) > 0$. Suppose that $P$ has $s$ edges for some $s \in \mathbb{N}$ and fix a linear ordering on these edges. We say that $m = (m_{i,j}) \in \mathbb{N}^{s \times k}$ is admissible for a word $w \in \Sigma^*$ if there exist $k$ distinct accepting runs of $P$ on $w$ such that $m_{i,j}$ is the number of times that the $i$-th edge is taken in the $j$-th accepting run.

For any ideal $I \subseteq \mathbb{N}^{s \times k}$ the set of $w \in \Sigma^*$ such that some $m \in I$ is admissible for $w$ is a regular language. A non-deterministic automaton for this language guesses $k$ distinct accepting runs of $P$ and counts the number of times each edge is taken on each accepting run, up to a finite threshold $N$, where $N$ is the largest integer appearing in the description of $I$ in the form (1). It follows that for any downward closed subset $D \subseteq \mathbb{N}^{s \times k}$ the set of $w \in \Sigma^*$ such that some $m \in D$ is admissible for $w$ is a regular language.

Now let $\lambda_1, \ldots, \lambda_s$ be the transition probabilities occurring in $P$, listed according to the ordering on the edges. Given $k \in \mathbb{N}$, consider the set of tuples

$$S_k = \left\{ (m_{i,j}) \in \mathbb{N}^{s \times k} : \sum_{j=1}^{k} \lambda_1^{m_{1,j}} \ldots \lambda_s^{m_{s,j}} > c \right\}.$$  

For any word $w \in \Sigma^*$, $w \in L_{\leq c}(P)$ if and only if there exists some $k$ up to the (finite) degree of ambiguity of $P$ and some $m \in S_k$ that is admissible for $w$. Since each set $S_k$ is downwards closed, it follows that $L_{\leq c}(P)$ is regular. ▶

The threshold language $L_{\leq c}(P)$ of a finitely ambiguous probabilistic automaton is regular, however, this does not say anything on how to decide efficiently whether $L_{\leq c}(P)$ is empty or not. Therefore, the next step is to bound the size of a witness word whenever $L_{\leq c}(P) \neq \emptyset$. This will lead to upper bounds on the complexity of the emptiness problem restricted to $k$-ambiguous.

**Lemma 4.** Let $P$ be a $k$-ambiguous probabilistic automaton with $n$ states. For every word $w$, there exists a word $w'$ of length at most $n^k$ such that $P(w) \preceq P(w')$.

This implies that the value of $P$ is reached by some word of length at most $n^k$.  

---

> 19:6  
> Probabilistic Automata of Bounded Ambiguity  
>  
> a finitely ambiguous probabilistic automaton $P$ and threshold $c$. A classical result, due to Rabin [16], shows that the threshold languages need not be regular in general. Unfortunately the proof of regularity, while constructive, is not useful for determining the complexity of the emptiness problem. However we are able to give a direct simple argument that bounds the length of witnesses for the emptiness problem. We then use these bounds to analyse the complexity of the emptiness problem.  
>  
> **Theorem 3.** Let $P$ be a finitely ambiguous probabilistic automaton and $c$ a threshold. Then $L_{\leq c}(P)$ is a regular language.  
>  
> **Proof.** Consider the set $\mathbb{N}^k$ under the pointwise order. Recall that $I \subseteq \mathbb{N}^k$ is an ideal if it is downward closed and directed. Every ideal $I$ has the form  
>  
> $$I = \{(n_1, \ldots, n_k) \in \mathbb{N}^k : n_{i_1} \leq a_1 \land \ldots \land n_{i_s} \leq a_s\}$$  
>  
> (1)  
>  
> for certain indices $1 \leq i_1 < \ldots < i_s \leq k$ and natural numbers $a_1, \ldots, a_s$. From the fact that $\mathbb{N}^k$ is a well-quasi-order it follows that every downward closed subset of $D \subseteq \mathbb{N}^k$ can be written as a finite union of ideals. Indeed such a decomposition can easily be computed from the finite set of minimal elements of $\mathbb{N}^k \setminus D$ [12].  
>  
> Let $P = (Q, q_0, \Delta, F)$ be a finitely ambiguous probabilistic automaton with transition function $\Delta : Q \times \Sigma \to D(Q)$. Say that a triple $(p, a, q) \in Q \times \Sigma \times Q$ is an edge if $\Delta(p, q)(a) > 0$. Suppose that $P$ has $s$ edges for some $s \in \mathbb{N}$ and fix a linear ordering on these edges. We say that $m = (m_{i,j}) \in \mathbb{N}^{s \times k}$ is admissible for a word $w \in \Sigma^*$ if there exist $k$ distinct accepting runs of $P$ on $w$ such that $m_{i,j}$ is the number of times that the $i$-th edge is taken in the $j$-th accepting run.  
>  
> For any ideal $I \subseteq \mathbb{N}^{s \times k}$ the set of $w \in \Sigma^*$ such that some $m \in I$ is admissible for $w$ is a regular language. A non-deterministic automaton for this language guesses $k$ distinct accepting runs of $P$ and counts the number of times each edge is taken on each accepting run, up to a finite threshold $N$, where $N$ is the largest integer appearing in the description of $I$ in the form (1). It follows that for any downward closed subset $D \subseteq \mathbb{N}^{s \times k}$ the set of $w \in \Sigma^*$ such that some $m \in D$ is admissible for $w$ is a regular language.  
>  
> Now let $\lambda_1, \ldots, \lambda_s$ be the transition probabilities occurring in $P$, listed according to the ordering on the edges. Given $k \in \mathbb{N}$, consider the set of tuples  
>  
> $$S_k = \left\{ (m_{i,j}) \in \mathbb{N}^{s \times k} : \sum_{j=1}^{k} \lambda_1^{m_{1,j}} \ldots \lambda_s^{m_{s,j}} > c \right\}.$$  
>  
> For any word $w \in \Sigma^*$, $w \in L_{\leq c}(P)$ if and only if there exists some $k$ up to the (finite) degree of ambiguity of $P$ and some $m \in S_k$ that is admissible for $w$. Since each set $S_k$ is downwards closed, it follows that $L_{\leq c}(P)$ is regular. ▶
>  
> The threshold language $L_{\leq c}(P)$ of a finitely ambiguous probabilistic automaton is regular, however, this does not say anything on how to decide efficiently whether $L_{\leq c}(P)$ is empty or not. Therefore, the next step is to bound the size of a witness word whenever $L_{\leq c}(P) \neq \emptyset$. This will lead to upper bounds on the complexity of the emptiness problem restricted to $k$-ambiguous.  
>  
> **Lemma 4.** Let $P$ be a $k$-ambiguous probabilistic automaton with $n$ states. For every word $w$, there exists a word $w'$ of length at most $n^k$ such that $P(w) \preceq P(w')$.  
>  
> This implies that the value of $P$ is reached by some word of length at most $n^k$.
Proof. Let \( \mathcal{P} = (Q, q_0, \Delta, F) \) and suppose that there are exactly \( k' \) accepting runs on \( w \) for some \( k' \leq k \). If \( w \) has length strictly greater than \( n^{k'} \) then there exists a factorization \( w = xyz \) for \( x, y, z \in \Sigma^* \), with \( y \) non-empty and \( xz \) of length at most \( n^k \), such that for each of the accepting runs on \( w \), the infix corresponding to the factor \( y \) starts and ends in the same state. Then we have

\[
\mathcal{P}(w) = \sum_{q \in F} \sum_{p \in Q} \mathcal{P}(q_{in} \xrightarrow{y} p) \mathcal{P}(p \xrightarrow{z} p) \mathcal{P}(p \xrightarrow{q}) \\
\leq \sum_{q \in F} \sum_{p \in Q} \mathcal{P}(q_{in} \xrightarrow{x} p) \mathcal{P}(p \xrightarrow{z} q) \\
= \mathcal{P}(xz).
\]

Note that if \( k \) is fixed, then the size of a witness for \( L^{>c}(\mathcal{P}) \) is polynomial in the size of the automaton (i.e. \( n^k \) where \( n \) is the number of states of \( \mathcal{P} \)). Unfortunately, it has been shown in [18] that the ambiguity of a finitely ambiguous automata can be exponential in the number of states and, thus, the previous lemma gives a double exponential bound for a witness of \( L^{>c}(\mathcal{P}) \) when \( k \) is not fixed. The next result shows that the size of a witness is at most exponential in the number of states.

**Theorem 5.** Let \( \mathcal{P} \) be a finitely ambiguous probabilistic automaton with \( n \) states. For every word \( w \), there exists a word \( w' \) of length at most \( n! \) such that \( \mathcal{P}(w) \leq \mathcal{P}(w') \).

This implies that the value of \( \mathcal{P} \) is reached by some word of length at most \( n! \).

Proof. Consider a word \( w = a_1 \cdots a_k \) of length at least \( n! \). For any position \( i \) over the runs of \( \mathcal{P} \) over \( w \), denote \( R_i \) the set of states participating in at least one accepting run over \( w \). Furthermore, we equip \( R_i \) with the order defined by \( p \leq q \) if \( \mathcal{P}(q_0 \xrightarrow{a_1 \cdots a_i} p) \leq \mathcal{P}(q_0 \xrightarrow{a_1 \cdots a_i} q) \), i.e. the probability of reading the prefix of \( w \) until position \( i \) reaches \( p \) with smaller probability than \( q \) (ties are resolved in a consistent way).

Since \( w \) has length at least \( n! \), there exist two positions \( i < j \) such that the ordered sets \( R_i \) and \( R_j \) coincide, denoted by \( R \), and there exists a factorization \( w = xyz \), with \( y \) the word in between positions \( i \) and \( j \). Then we look at the runs of \( y \) from \( R \) to \( R \), and make the following claims:

1. For every \( p \in R \), there exists a run over \( y \) from \( p \) to a state in \( R \).
2. For every \( p \in R \), there exists at most one run over \( y \) from \( p \) to a state in \( R \).
3. For every \( p \in R \), we have \( \mathcal{P}(q_0 \xrightarrow{a_1 \cdots a_i} p) \leq \mathcal{P}(q_0 \xrightarrow{a_1 \cdots a_j} p) \).

The first claim follows from the fact that \( R \) is the set of states participating in at least one accepting run over \( w \). For the second claim, if this would not be the case, then the number of runs from \( R \) to \( R \) would increase unboundedly, contradicting that \( \mathcal{P} \) is finitely ambiguous. Then it follows that for any state \( p \in R \) there exists a unique run over \( y \) from \( p \) to some state in \( R \), which is written \( p' \).

For the last item, pick a state \( p \in R \) and note that \( \mathcal{P}(q_0 \xrightarrow{xy} p') = \mathcal{P}(q_0 \xrightarrow{x} p) \cdot \mathcal{P}(p \xrightarrow{y} p') \leq \mathcal{P}(q_0 \xrightarrow{z} p) \). This reduce the analysis to two cases. On one hand, \( p \leq p' \) and then \( \mathcal{P}(q_0 \xrightarrow{xy} p) \leq \mathcal{P}(q_0 \xrightarrow{xy} p') \leq \mathcal{P}(q_0 \xrightarrow{x} p) \). On the other hand, \( p > p' \) and then there exists a state \( q \) in \( R \) such that \( q \leq p \) and \( p \leq q' \). This is because for any state \( r \in R \) there exists a unique run over \( y \) from \( r \) to some state in \( R \). It follows that \( \mathcal{P}(q_0 \xrightarrow{x} p) \leq \mathcal{P}(q_0 \xrightarrow{xy} q') \leq \mathcal{P}(q_0 \xrightarrow{z} q) \leq \mathcal{P}(q_0 \xrightarrow{z} p) \).

Finally, the proof of the theorem follows from the last claim (see Lemma 4).

With the previous bounds in hand, we can study the computational complexity of the emptiness problem for various classes of finitely ambiguous probabilistic automata. For each fixed positive integer \( k \) we consider the class of \( k \)-ambiguous probabilistic automata. More
generally, we can let the ambiguity of an automaton depend on the number \( n \) of states: we consider for each fixed polynomial \( p \) the class of all automata that have ambiguity at most \( p(n) \). We call this the class of automata of \( p \)-bounded ambiguity. More generally still, we have the class of all finitely ambiguous probabilistic automata. (Recall that the ambiguity can be exponential in the number of states in general.)

\[\textbf{Theorem 6.}\]

- For each fixed positive integer \( k \), the emptiness problem for the class of \( k \)-ambiguous probabilistic automata is in \( \text{NP} \).
- For each fixed polynomial \( p \), the emptiness problem for the class of probabilistic automata with \( p \)-bounded ambiguity is in \( \text{PSPACE} \). This problem is \text{PSPACE}-hard already in case \( p(n) = n \).
- The emptiness problem for the class of finitely ambiguous probabilistic automata is in \( \text{NEXPTIME} \) and is \text{PSPACE}-hard.

\[\textbf{Proof.}\] The algorithm for all three cases exploits Lemma 4 and Theorem 5 to guess and check a word witnessing that threshold language is non-empty.

For \( k \)-ambiguous we know by Lemma 4 that a witness for checking whether \( L^{>c}(\mathcal{P}) \neq \emptyset \) is of polynomial size in \( \mathcal{P} \) and, therefore, we can guess a polynomial size word \( w \) and check if \( \mathcal{P}(w) \geq c \), that is, the problem is in \( \text{NP} \).

Similarly, for finitely ambiguous we know by Theorem 5 that the witness is of size at most exponential, so we can guess and check if \( L^{>c}(\mathcal{P}) \) is non-empty in \( \text{NEXPTIME} \).

To show that emptiness is in \( \text{PSPACE} \) for probabilistic automata of \( p \)-bounded ambiguity, one can guess a word \( w \) “on the fly” of size exponential and check whether \( \mathcal{P}(w) \geq c \). The problem here is that the value \( \mathcal{P}(w) \) (written in binary) could be of size exponential in the size of \( \mathcal{P} \). To check if \( \mathcal{P}(w) \geq c \) with polynomial space one can guess \( w \), and keep a set of counters \( \{c_i\} \) that stores how many times each transition \( t \) is used on the \( i \)-th run of \( \mathcal{P} \) over \( w \). Since \( w \) is of size at most exponential and \( \mathcal{P} \) has at most \( p(n) \) accepting runs, then we need polynomially many counters, each with at most polynomially many bits, namely, polynomial space to store these counters during the simulation of \( \mathcal{P} \) over \( w \). After we conclude guessing \( w \), we can construct a polynomial-size circuit that receives \( \{c_i\} \) and outputs \( \mathcal{P}(w) \). Checking that the value of the circuit is greater or equal than a constant \( c \) correspond to decide \text{PosSLP} which can be solved in \( \text{PSPACE} \) [1].

Next we consider a fixed polynomial \( p(n) = n \), and prove \text{PSPACE}-hardness of emptiness for the class of probabilistic automata of \( p(n) \)-bounded ambiguity. The proof is by reduction from the emptiness problem of the intersection of a finite collection of deterministic finite automata: given as input a collection of deterministic finite automata, does there exist a word accepted by each of them? This problem has been shown \text{PSPACE}-complete in [11]. Given \( N \) deterministic automata, we construct a probabilistic automaton \( \mathcal{P} \) whose first letter leads with probability \( \frac{1}{N} \) to the initial state of each automaton. The probabilistic automaton \( \mathcal{P} \) is \( N \)-ambiguous (note that \( N \) is at most the number of states of \( \mathcal{P} \)), and there exists a word \( w \) such that \( \mathcal{P}(w) = 1 \) if, and only if, there exists a word accepted by each of the \( N \) deterministic automata.

The aim of the last section is to give better algorithms for the \( k \)-ambiguous case: in particular, we show that the emptiness problem is in quasi polynomial time for 2-ambiguous probabilistic automata.
5 Algorithms and Approximations for Finitely Ambiguous Probabilistic Automata

This section is devoted to the construction of algorithms for both the emptiness problem and approximating the value of finitely ambiguous probabilistic automata. The first step is a reduction to a multi-objective optimisation problem that we call the stochastic path problem.

We construct algorithms for this problem relying on recent progress on the literature of multi-objective optimisation problems, and thus obtain algorithms for finitely ambiguous probabilistic automata.

5.1 The Stochastic Path Problem

The stochastic path problem is an optimisation problem on multi-weighted graphs. It is parametrised by a positive integer constant \( k \), giving rise to the \( k \)-stochastic path problem, and the bi-stochastic path problem for \( k = 2 \). An instance is a triple consisting of an acyclic \( k \)-weighted graph \( G \) and two vertices \( s \) and \( t \). A \( k \)-weighted graph is given by a set of vertices \( V \) of size \( n \) and a set of weighted edges \( E \subseteq V \times (\mathbb{Q} \cap [0,1])^k \times V \). Note that the same pair of vertices \( (v,v') \) can have different edges between them and the weight of an edge is a \( k \)-tuple of rational numbers between 0 and 1.

A path \( \pi \) in \( G \) is a sequence of consecutive edges, and the set of feasible solutions of the problem are all paths from \( s \) to \( t \). We denote by \( \left( p_1(\pi), \ldots, p_k(\pi) \right) \) the component-wise product of the weight vectors along the edges of \( \pi \), namely, weights are computed multiplicatively along each component. In our applications we think of each component of a weight vector of an edge as the probability of a single event, and each component of a weight vector of a path as the probability of a sequence of events. The value of the path \( \pi \), denoted by \( \text{val}(\pi) \), is obtained by summing each component of the weight vector of the path, namely, \( \text{val}(\pi) = \sum_{i=1}^{k} p_i(\pi) \).

As a running example, on the left-hand side of Figure 2 we represent an instance of the bi-stochastic path problem. There are five paths from \( s \) to \( t \), and their values are plotted in the right-hand side. For instance, the path \( s, p, q, t \) using the left edge from \( p \) to \( q \) has weight \( (.4 \times .9 \times .9, .6 \times .1 \times .9) = (.324, .054) \).

The objective of the \( k \)-stochastic path problem is to maximize the sum of the objective functions, namely, \( \text{val}(\pi) \). The value of the above path is \(.324 + .054 = .378 \). Finally, the decision problem associated with the \( k \)-stochastic path problem is the following:

\[ \text{The } k \text{-stochastic path problem: given a } k \text{-weighted graph } G \text{, two vertices } s \text{ and } t \text{ and a threshold } c \in \mathbb{Q} \cap [0,1], \text{ does there exist a path } \pi \text{ from } s \text{ to } t \text{ in } G \text{ whose value is at least } c, \text{ i.e. such that } \text{val}(\pi) \geq c? \]

Towards finding efficient algorithms and approximations of \( k \)-ambiguous probabilistic automata, we show a polynomial time reduction from the emptiness problem of \( k \)-ambiguous probabilistic automata to the \( k \)-stochastic path problem. Intuitively, the reduction consists in constructing the powerset graph of the paths, restricting to at most \( k \) paths.

Lemma 7. The emptiness problem of a \( k \)-ambiguous probabilistic automaton \( P \) reduces in polynomial time to a \( k \)-stochastic path problem \((G_P, s, t)\). In particular, the reduction satisfies that

1. for any word \( w \) there exists a path \( \pi \) in \( G \) from \( s \) to \( t \) such that \( P(w) \leq \text{val}(\pi) \),
2. for any path \( \pi \) in \( G \) from \( s \) to \( t \) there exists a word \( w \) such that \( \text{val}(\pi) \leq P(w) \).
Figure 2 An instance of the bi-stochastic path problem on the left, and the values of all paths from s to t on the right. The four red dots are the Pareto curve, and the three connected red dots the convex Pareto curve.

Proof. Let $\mathcal{P} = (Q, q_{in}, \Delta, F)$ be a $k$-ambiguous probabilistic automaton with $n$ states. The set of vertices of the $k$-weighted graph $G_\mathcal{P}$ is defined as $Q^k \times \{0, \ldots, n^k\} \times \{0, 1\}^{k \times k}$ where $\{0, 1\}^{k \times k}$ is the set of $k \times k$ matrices over $\{0, 1\}$, plus a special source vertex $s$ and a special target vertex $t$.

Intuitively, being in the vertex $((q_1, \ldots, q_k), \ell, M)$ means that we are simulating $k$ runs which are now in the states $(q_1, \ldots, q_k)$, that the run so far has length $\ell$, and the matrix $M$ indicates which of two $k$ runs are different: $M(i, j) = 1$ if, and only if, the $i$-th run is different from the $j$-th run.

The set of edges is defined accordingly to the previous explanation as follows. For the source vertex, there is an edge from $s$ to $((q_{in}, \ldots, q_{in_{n}}), 0, 0)$ with weight $(1, \ldots, 1)$, where 0 is the zero matrix. There is an edge from $((q_1, \ldots, q_k), \ell, M)$ to $((q'_1, \ldots, q'_k), \ell + 1, M')$ with weight $(p_1, \ldots, p_k)$ if there exists a letter $a$ in $\Sigma$ such that for each $i \in \{1, \ldots, k\}$ we have $\Delta(q_i, a)(q'_i) = p_i$, and $M'(i, j) = 1$ if, and only if, $M(i, j) = 1$ or $q'_i \neq q'_j$. Finally, there is an edge from $((q_1, \ldots, q_k), \ell, M)$ to $t$ with weight $(p_1, \ldots, p_k)$ where for each $i \in \{1, \ldots, k\}$ we have $p_i = 1$ if $q_i \in F$ and $M(i, j) = 1$ for every $j < i$, and $p_i = 0$ otherwise. Note that $G_\mathcal{P}$ is acyclic and of polynomial size given that $k$ is a fixed value.

We prove the correctness of the construction. Let $w$ be a word. Thanks to Lemma 4, we can assume without loss of generality that $w$ has length at most $n^k$. Its set of accepting runs induces a path $\pi$ in $G_\mathcal{P}$ from $s$ to $t$ with $\text{val}(\pi) = \mathcal{P}(w)$. Conversely, a path $\pi$ in $G_\mathcal{P}$ from $s$ to $t$ corresponds to a set of accepting runs for some word $w$ with $\text{val}(\pi) \leq \mathcal{P}(w)$. □

By the previous result, we can see that the emptiness problem of $k$-ambiguous probabilistic automata is closely related to the $k$-stochastic path problem. In the following we use this problem as a proxy to give approximation and efficient algorithms for the emptiness problem.

5.2 Approximating the Value in Polynomial Time

Multi-objective optimisation problems have long been studied; see Papadimitriou and Yannakakis [14] and Diakonikolas and Yannakakis [6] among many others. Since there is typically no single best solution, a natural notion for multi-objective optimisation problems is Pareto curves, which is the set of undominated solutions. To make things concrete, we illustrate the notion of Pareto curves on the $k$-stochastic path problem. We fix an instance $(G, s, t)$ of the $k$-stochastic path problem. A Pareto curve is a set of paths $\mathcal{P}$ such that for every path $\pi$, there exists a path $\pi'$ in $\mathcal{P}$ dominating $\pi$, i.e. such that for all $i \in \{1, \ldots, k\}$, we have
\( p_i(\pi) \leq p_i(\pi') \). In Figure 2, we can see that the Pareto curve of our running example is given by the four red dots. Here dominating means being to the right and higher, so only one path is dominated by others. Unfortunately, the size of Pareto curves in discrete multi-objective optimisation problems is exponential in the worst case. Hence the introduction of two relaxations: convex and approximate Pareto curves.

A **convex Pareto curve** is a set of paths \( C \) such that for every path \( \pi \), there exists a family of paths \( \pi_1, \ldots, \pi_m \in C \) such that \( \pi \) is dominated by a convex combination of \( \pi_1, \ldots, \pi_m \) in the sense that there exist non-negative coefficients \( \lambda_1, \ldots, \lambda_m \) that sum to 1 such that \( p_i(\pi) \leq \sum_j \lambda_j p_i(\pi_j) \) for all components \( i \) in \( \{1, \ldots, k\} \).

Convex Pareto curves have been studied in a general setting by Diakonikolas and Yannakakis [14]. They are in general smaller than Pareto curves, yielding efficient algorithms for convex optimisation problems.

In Figure 2, there exists a convex Pareto curve consisting of only three paths, the fourth one being dominated a convex combination of two other paths. The figure connects the three dots, showing which is sometimes called the Pareto front.

Fix \( \varepsilon > 0 \), an \( \varepsilon \)-**Pareto curve** is a set of paths \( C \) such that for every path \( \pi \), there exists a path \( \pi' \) in \( C \) such that for all \( i \) in \( \{1, \ldots, k\} \), we have \( p_i(\pi) \leq (1 + \varepsilon) \cdot p_i(\pi') \).

The notion of approximate Pareto curves is very appealing in our case for two reasons: first, knowing an approximate Pareto curve usually gives an approximately optimal solution, and second, a very general result of Papadimitriou and Yannakakis [14] shows that in most multi-objective optimisation problems, there exists a polynomially succinct approximate Pareto curve.

The two relaxations are combined to give rise to the notion of \( \varepsilon \)-**convex Pareto curves**: it is a set of paths \( C \) such that for every path \( \pi \), there exists a family of paths \( \pi_1, \ldots, \pi_m \in C \) such that there exist non-negative coefficients \( \lambda_1, \ldots, \lambda_m \) that sum to 1 such that \( p_i(\pi) \leq (1 + \varepsilon) \sum_j \lambda_j p_i(\pi_j) \) for all components \( i \) in \( \{1, \ldots, k\} \).

The following result shows the way to find an \( \varepsilon \)-approximation of the value of a \( k \)-ambiguous probabilistic automaton \( \mathcal{P} \).

**Theorem 8.** For any fixed \( k \), there exists a polynomial time algorithm which given an instance of the \( k \)-stochastic path problem and \( \varepsilon > 0 \), returns an \( \varepsilon \)-convex Pareto curve in time polynomial in the instance and \( \frac{1}{\varepsilon} \).

**Proof.** We rely on general results of Papadimitriou and Yannakakis [14], which give a sufficient condition for the existence of a polynomial time algorithm constructing an \( \varepsilon \)-convex Pareto curve in time polynomial in the instance and \( \frac{1}{\varepsilon} \): it is enough to construct a pseudo-polynomial time algorithm solving the exact version in pseudo-polynomial time. Recall here that an algorithm is pseudo-polynomial if it runs in polynomial time when the numerical inputs are given in unary.

In our case, the exact \( k \)-stochastic path problem reads: given an instance \( (G, s, t) \) and a value \( c \) in \( [0, 1] \cap \mathbb{Q} \), does there exist a path \( \pi \) in \( G \) from \( s \) to \( t \) such that \( \sum_{i \in \{1, \ldots, k\}} p_i(\pi) = c \)? If all transition probabilities are written using \( B \) bits, then it is enough to consider paths such that each weight uses at most \( |V| \cdot B \) bits with \( V \) the set of vertices of \( G \). Hence one can fill in a polynomially large table indexed by \( (p, q, p_1, \ldots, p_k) \), which checks for the existence of a path from \( p \) to \( q \) of weights \( (p_1, \ldots, p_k) \) using at most \( |V| \cdot B \) bits. ▶
Interestingly, the algorithm of Theorem 8 for the $k$-stochastic path problem yields a polynomial time algorithm to approximate the value of a $k$-ambiguous probabilistic automaton. Recall that the value of a probabilistic automaton $P$ is defined by $\text{val}(P) = \sup_{w \in \Sigma^*} P(w)$.

> **Theorem 9.** For a fixed $k$, there exists an algorithm which given a $k$-ambiguous probabilistic automaton and $\varepsilon > 0$, outputs an $\varepsilon$-approximation of the value in time polynomial in the size of the automaton and $\frac{1}{\varepsilon}$, i.e. a value $\text{Output}$ such that

$\text{Output} \leq \text{val}(P) \leq (1 + \varepsilon) \cdot \text{Output}$.

**Proof.** Given a $k$-ambiguous probabilistic automaton $P$, the algorithm for finding an $\varepsilon$-approximation of $\text{val}(P)$ is as follows:

1. construct an instance $(G_P, s, t)$ of the $k$-stochastic path problem using Lemma 7.
2. construct an $\varepsilon$-convex Pareto curve $C$ for $(G_P, s, t)$ thanks to Theorem 8.
3. output $\text{Output} = \max_{\pi \in C} \sum_{i \in \{1, \ldots, k\}} p_i(\pi)$.

A direct application of Lemma 7 shows that $\text{Output}$ is an $\varepsilon$-approximation of $\text{val}(P)$.

Can we approximate the value of any finitely ambiguous probabilistic automaton in polynomial time? Unfortunately, by reformulating the hardness result of [13] (that paper uses a different framework), we can give a negative answer to this question, justifying the relevance of Theorem 9.

> **Theorem 10 ([13]).** For every $\varepsilon > 0$, there is no polynomial time algorithm computing the value of finitely ambiguous probabilistic automata up to a factor $O(n^{\frac{1}{2} - \varepsilon})$ unless $P = \text{NP}$.

The paper [13] constructs a reduction from the size of the maximum clique, for which we know that no polynomial time approximation algorithm exists unless $P = \text{NP}$. The construction is given in [13] for Hidden Markov models; we illustrate in Figure 3 how to adapt it to probabilistic automata.

Given a graph $G$ with $n$ vertices, we construct a finitely ambiguous probabilistic automaton $P$ with $n^2$ states such that for each $m$ smaller than $n$, the automaton accepts a word with probability at least $\frac{m}{2n^2}$ if, and only if, the graph contains a clique of size at least $m$.

We give an intuitive explanation for the construction. A word over $\{a, b\}^*$ represents a set of vertices in the graph; $a$ means in the set and $b$ outside. For instance, the word $aaab$...
represents the set of vertices \{v_1, v_2, v_3\}, it has probability \(\frac{3}{4} \cdot \frac{2}{3}\). The automaton \(\mathcal{P}\) on input \(w\) has \(n\) runs, one for each vertex, chosen each with probability \(\frac{1}{n}\). Each accepting run has probability \(\frac{1}{2n-1}\), and the run corresponding to a vertex \(v\) is successful if, and only if, \(v\) and all its neighbours belong to the set of vertices represented by the word \(w\). Hence a clique of size \(m\) induces a word accepted with probability \(\frac{m}{2n^{2n-1}}\), and conversely.

Let \(\text{MaxClique}(G)\) denote the size of the largest clique in \(G\), the equivalence above reads \(\text{MaxClique}(G) = \frac{n}{2n^{2n-1}} \cdot \text{val}(\mathcal{P})\). It follows that a \(K(n)\)-approximation algorithm for the value of finitely ambiguous probabilistic automata induces a \(K(n^2)\)-approximation algorithm for the size of the largest clique. It has been proved that \(\text{MaxClique}(G)\) cannot be approximated within a factor better than \(O(n^{1-\epsilon})\) for every \(\epsilon > 0\), unless \(P = NP\) [19], implying our result.

### 5.3 A Quasi-Polynomial Time Algorithm for 2-ambiguous Probabilistic Automata

The previous results show that one can always \(\epsilon\)-approximate the value of \(k\)-ambiguous probabilistic automaton. This is however not enough to decide the emptiness problem. On this direction, Theorem 6 shows that for any fixed \(k\) the emptiness problem of \(k\)-ambiguous probabilistic automata is in \(NP\). We show that for \(k = 2\) there exists a quasi-polynomial time algorithm for the emptiness problem. For this, we start by showing a quasi-polynomial time algorithm for the bi-stochastic path problem.

\[\text{Theorem 11. There exists an algorithm which given an instance of the bi-stochastic path problem, returns a convex Pareto curve in quasi-polynomial time.}\]

The advantage of using \(k = 2\) relies on the existence of a quasi-polynomial bound on the size of convex Pareto curves. More precisely, if \((G, s, t)\) is an instance of the bi-stochastic path problem with \(n\) vertices, then it can be shown that there exists a convex Pareto curve of size at most \(n^{\log(n)}\). This result was proved by Gusfield in his PhD thesis [10], and a matching lower bound was developed by Carstensen [3]. Note that they use a different framework, called parametric optimisation: in the parametric shortest path problem each edge has cost \(c + \lambda d\), where \(\lambda\) is a parameter. The length of the shortest path is a piecewise linear concave function of \(\lambda\), whose pieces correspond to the vertices of the convex Pareto curve for the bi-objective shortest path problem with weights \((c, d)\). It is then easy to obtain an upper bound on the size of convex Pareto curves for the bi-stochastic path problem by reducing it to a bi-objective shortest path problem, mapping the weights \((p, q)\) to \((-\log(p), -\log(q))\). Finally, the upper bound on the size of convex Pareto curves yields a quasi-polynomial time algorithm, by constructing them in a standard divide-and-conquer manner.

The algorithm of Theorem 11 yields a quasi-polynomial time algorithm for the emptiness problem of 2-ambiguous probabilistic automata.

\[\text{Theorem 12. There exists a quasi-polynomial time algorithm for the emptiness problem of 2-ambiguous probabilistic automata.}\]

\[\text{Proof. Given a 2-ambiguous probabilistic automaton } \mathcal{P} \text{ and a threshold } c, \text{ the algorithm for deciding the emptiness of } \mathcal{P} \text{ is the following:}\]

- construct an instance \((G_P, s, t)\) of the bi-stochastic path problem using Lemma 7.
- construct a convex Pareto curve \(C\) for \((G_P, s, t)\) thanks to Theorem 11.
- check whether \(\text{Output} = \max_{\pi \in C} \sum_{i} p_i(\pi) > c\).

A direct application of Lemma 7 shows that \(\text{Output} = \text{val}(\mathcal{P})\).

We do not know whether there exist quasi-polynomial time algorithms for every \(k > 2\), and leave this as an open problem.
References

Two Lower Bounds for BPA∗†

Mingzhang Huang1 and Qiang Yin2

1 BASICS, Shanghai Jiao Tong University, Shanghai, China
mingzhanghuang@gmail.com
2 BDBC, Beihang University, Beijing, China
yinqiang@buaa.edu.cn

Abstract
Branching bisimilarity of normed Basic Process Algebra (nBPA) was claimed to be EXPTIME-hard in previous papers without any explicit proof. Recently it has been pointed out by Petr Jančar that the claim lacked proper justification. In this paper, we develop a new complete proof for the EXPTIME-hardness of branching bisimilarity of nBPA. We also prove that the associated regularity problem of nBPA is PSPACE-hard. This improves previous P-hard result.

1998 ACM Subject Classification F.4.2 Grammars and Other Rewriting Systems

Keywords and phrases BPA, branching bisimulation, regularity

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2017.20

1 Introduction
Equivalence checking is a core issue of verification. It asks whether two systems are related by a specific equivalence. Baeten, Bergstra and Klop proved a remarkable result that strong bisimilarity between context free grammars without empty production is decidable [1]. This result was surprising because it seems to contradict the well-known fact that language equivalence between these grammars is undecidable. Context free grammars without empty production can be regarded as normed Basic Process Algebra (nBPA) processes. Moreover, their work extended the decidability result of bisimulation equivalence from finite state systems to infinite state systems. Extensive work has appeared since their inspiring paper, dealing with decidability or complexity issues of checking bisimulation equivalences on various infinite state systems (see a survey [11] and an updated overview [18] on this topic).

The decidability of weak bisimilarity over BPA is one of the central open problems. Although this problem is generally believed to be decidable, so far there is no effective method to handle the difficulties caused by arbitrary silent transitions. We do not know if the weak bisimilarity is decidable or not even for normed BPA. Mayr showed the weak bisimilarity and regularity problem of general BPA are EXPTIME-hard [13]. The regularity problem asks if a BPA process is bisimilar to some (unspecific) finite state process. It is noteworthy that Kiefer showed that the strong bisimilarity problem is already EXPTIME-hard by constructing a reduction from the Hit-or-Run game [10].

The normed case seems easier than the general one. Fu proved that branching bisimilarity, a standard refinement of weak bisimilarity, is decidable on nBPA [5]. He also extended the decidability result to the associated regularity problem. Recently Czerwiński and Jančar

∗ This work was done in part while Qiang Yin was studying at BASICS, Shanghai Jiao Tong University. The full version is available at http://basics.sjtu.edu.cn/~yin.
† Minzhang Huang is supported in part by NSFC (61472239, ANR 61261130589). Qiang Yin is supported in part by Beijing Advanced Innovation Center for Big Data and Brain Computing, Beihang University.

© Mingzhang Huang and Qiang Yin; licensed under Creative Commons License CC-BY
28th International Conference on Concurrency Theory (CONCUR 2017).
Editors: Roland Meyer and Uwe Nestmann; Article No. 20; pp. 20:1–20:16
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany
improved both decidability results to NEXPTIME [4]. He and Huang further showed the branching bisimilarity of nBPA can actually be decided in EXPTIME [6]. However, for the weak bisimilarity of nBPA, there is no upper bound. Stříbrná first gave an NP-hard result by reducing from the Knapsack Problem [20]; Srba then improved it to PSPACE-hard by reducing from QSAT (Quantified SAT) [14]; and the best known lower bound is EXPTIME-hard given by Mayr by a reduction from the acceptance problem of alternating linear-bounded automaton (ALBA) [13]. A natural question is whether some of these lower bounds proofs hold for the branching bisimilarity of nBPA.

The branching bisimilarity of nBPA was claimed to be EXPTIME-hard [5, 6]. Researchers believed that some modifications on Mayr’s reduction [13] would transform the weak bisimilarity version reduction into a branching bisimilarity version reduction. However recently Jančar reminded that the claim lacked proper justification [7]. A typical modification of adding silent transition loop to make state bisimilar does not work. Unfortunately for the EXPTIME-hardness, almost any small modification is not a proper reduction from ALBA to the branching bisimilarity. According to the EXPTIME algorithm [6], we know that the main exponential factor is to do with the fact there are exponentially many redundant sets. The redundant set of a process consists of redundant variables that, when placed as a prefix to the process, gives rise to an equivalent process. In Mayr’s construction [13], the number of redundant sets is only polynomial. So that the algorithm might perform better under the input of particular constructions. The above modification method does not even work for the NP-hard [20] or PSPACE-hard [14] lower bound construction. The current lower bound of the branching bisimilarity of nBPA is merely P-hard [2]. The same happens to the regularity checking problem of nBPA. The only known lower bound for branching regularity is P-hard [2, 17]. Comparatively, the weak regularity problem is PSPACE-hard [14, 17].

Our Contribution. In this paper we study the lower bounds of the branching bisimilarity and branching regularity problems of normed BPA. The EXPTIME algorithm [6] hints that exponentially many redundant sets lead to exponential running time. We first introduce a novel way to design a structure with exponentially many of redundant sets. Then we use this structure to implement a binary counter and construct a reduction from the Hit-or-Run game [10]. This confirms the EXPTIME-hard lower bound for the branching bisimilarity of nBPA. We also present another reduction from QSAT to branching bisimilarity of nBPA by this structure. Combining with the Srba’s reduction from equivalence checking to regularity checking [17], we get a PSPACE-hard lower bound for the branching regularity of nBPA. Figure 1 summarizes the state of the art in equivalence checking and regularity checking with respect to weak and branching bisimilarity of nBPA. The results proved in this paper are marked in boldface.

Organization. Section 2 introduces some basic notions. Section 3 introduces the structure with exponentially many redundant sets. Section 4 proves the EXPTIME-hardness of the
equivalence checking. Section 5 proves the PSPACE-hard lower bound for the regularity checking. Section 6 concludes with some remarks.

\section{Preliminaries}

A BPA system $\Gamma$ is a tuple $(V, A, \Delta)$, where $V$ is a finite set of variables ranged over by $A, B, C, \ldots, X, Y, Z$; $A$ is a finite set of actions ranged over by $\lambda$; and $\Delta$ is a finite set of transition rules. We use a specific letter $\tau$ to denote internal action and use $a, b, c, d, e, f, g$ to range over visible actions from the set $A \setminus \{\tau\}$. A process defined in $\Gamma$ is a word $w \in V^*$. Processes is denoted by $\alpha, \beta, \gamma, \delta, \sigma$. The nil process is denoted by a special symbol $\epsilon$. We will use $=\!\!\!\!=$ for the syntactical equality, and $\alpha = \alpha = \alpha$ by convention. A rule in $\Delta$ is in the form $X \xrightarrow{\lambda} \alpha$, where $\alpha$ is a BPA process. The operational semantics of the processes is defined by the following rules.

\[
\begin{align*}
X &\xrightarrow{\lambda} \alpha \in \Delta & X \xrightarrow{\lambda} \alpha \quad &\alpha \xrightarrow{\lambda} \alpha' \quad &\alpha \beta \xrightarrow{\lambda} \alpha' \beta
\end{align*}
\]

We will write $\alpha \xrightarrow{*} \beta$ for $\alpha \xrightarrow{\cdot} \beta$ and $\xrightarrow{*}$ for the reflexive transitive closure of $\xrightarrow{\cdot}$.

A BPA process $\alpha$ is \textit{normed} if $\exists \lambda_1, \ldots, \lambda_k. \alpha \xrightarrow{\lambda_1} \ldots \xrightarrow{\lambda_k} \epsilon$. A BPA system is \textit{normed} if every variable is normed. We write $nBPA$ for normed BPA. We denote by $\mathcal{V}^BPA$ the set of variables in $V$ that can reach $\epsilon$ via internal actions alone. For a process $\alpha$, we use $\text{Var}(\alpha)$ to represent the set of variables occurring in $\alpha$; and we use $|\alpha|$ to denote the size of $\alpha$, which is defined to be the length of the word $\alpha$. The size of each rule $X \xrightarrow{\lambda} \alpha$ is defined to be $|\alpha| + 2$. The size of $\Delta$ is the sum of the size of all rules. The size of a BPA system $\Gamma = (V, A, \Delta)$ is defined by $|\Gamma| = |V| + |A| + |\Delta|$.

\textbf{Bisimulation Equivalence.} A symmetric relation $\mathcal{R}$ on BPA processes is a \textit{branching bisimulation} if whenever $\alpha \mathcal{R} \beta$ and $\alpha \xrightarrow{\lambda} \alpha'$ then one of the statements is valid:

\begin{itemize}
  \item $\lambda = \tau$ and $\alpha \mathcal{R} \beta$;
  \item $\beta \xrightarrow{*} \beta'' \xrightarrow{\lambda} \beta'$ for some $\beta'$ and $\beta''$ such that $\alpha \mathcal{R} \beta''$ and $\alpha' \mathcal{R} \beta'$.
\end{itemize}

If we replace the above second item by the following one

\begin{itemize}
  \item $\beta \xrightarrow{*} \gamma_1 \xrightarrow{\lambda} \gamma_2 \xrightarrow{*} \beta'$ for some $\gamma_1, \gamma_2$ and $\beta'$ such that $\alpha' \mathcal{R} \beta'$
\end{itemize}

then we get the definition of \textit{weak bisimulation}. The largest branching bisimulation, denoted by $\simeq$, is branching bisimilarity; and the largest weak bisimulation, denoted by $\approx$, is weak bisimilarity. It is obvious that both $\simeq$ and $\approx$ are equivalences and are congruences with respect to the composition operator in BPA model. Branching bisimilarity is a refinement of weak bisimilarity, \textit{i.e.} $\simeq \subseteq \approx$. We say $\alpha$ and $\beta$ are branching bisimilar (weak bisimilar) if $\alpha \simeq \beta$ ($\alpha \approx \beta$). Both branching and weak bisimilarity satisfy a standard property of observational equivalence stated as follows.

\textbf{Lemma (Computation Lemma).}

\begin{itemize}
  \item If $\alpha \xrightarrow{\cdot} \alpha_1 \xrightarrow{\cdot} \ldots \xrightarrow{\cdot} \alpha_k$ and $\alpha \simeq \alpha_k$, then for all $1 \leq i \leq k$ we have $\alpha \simeq \alpha_i$.
  \item If $\alpha \xrightarrow{\cdot} \alpha_1 \xrightarrow{\cdot} \ldots \xrightarrow{\cdot} \alpha_k$ and $\alpha \approx \alpha_k$, then for all $1 \leq i \leq k$ we have $\alpha \approx \alpha_i$.
\end{itemize}

\textbf{Bisimulation Game.} Bisimulation relation has a standard game characterization [21, 19] which is very useful for studying the lower bounds. A branching (resp. weak) bisimulation game is a 2-player game played by \textit{Attacker} and \textit{Defender}. A configuration of the game is pair of processes $(\alpha_0, \alpha_1)$. The game is played in rounds. Each round has 3 steps: (1)
Attacker chooses a move; (2) Defender responds to match Attacker’s move; (3) Attacker sets the next round configuration according to Defender’s response. One round of branching bisimulation game is defined as follows, assuming \((\beta_0, \beta_1)\) is the configuration of the current round.

1. Attacker picks up \(i \in \{0, 1\}\), \(\lambda\), and \(\beta_i'\) to play \(\beta_i \xrightarrow{\lambda} \beta_i'\).
2. Defender responds with \(\beta_{1-i} \xrightarrow{\lambda} \beta_{1-i}'\) for some \(\beta_{1-i}'\). Defender can also play an empty response when \(\lambda = \tau\) and we stipulate that \(\beta_{1-i}' = \beta_{1-i}\) if Defender plays an empty response.
3. Attacker sets \((\beta_i', \beta_{1-i}')\) as the configuration of the next round if Defender plays an empty response; otherwise Attacker sets either \((\beta_i', \beta_{1-i}')\) or \((\beta_i, \beta_{1-i})\) as the configuration of the next round.

A round of weak bisimulation game differs from the above one in the last 2 steps.

1. Defender responds with \(\beta_{1-i} \xrightarrow{\lambda} \beta_{1-i}'\) for some \(\beta_{1-i}'\); Defender can also play an empty response when \(\lambda = \tau\) and we stipulate that \(\beta_{1-i}' = \beta_{1-i}\) in that case.
2. Attacker sets the configuration of the next round to be \((\beta_i', \beta_{1-i}')\).

If one player gets stuck, the other one wins. If the game goes on for infinitely many rounds, then Defender wins. We say a player has a winning strategy, w.s. for short, if he or she can win no matter how the other one plays. Defender has a w.s. in the branching bisimulation game \((\alpha, \beta)\) iff \(\alpha \succeq \beta\); Defender has a w.s. in the weak bisimulation game \((\alpha, \beta)\) iff \(\alpha \approx \beta\).

**Redundant Set.** The concept of redundant set was first introduced by Fu [5] to show the decidability of branching bisimilarity of nBPA. It also plays an important role in the branching bisimilarity checking algorithms of nBPA [4, 6]. Given a nBPA system \(\Gamma = (\mathcal{V}, \mathcal{A}, \Delta)\), the redundant set of \(\alpha\), notation \(\text{RD}(\alpha)\), is the set of variables defined by

\[
\text{RD}(\alpha) = \left\{ X \in \mathcal{V} \mid X\alpha \succeq \alpha \right\}
\]

(1)

It is necessary that \(\text{RD}(\alpha) \subseteq \mathcal{V}^0\). Note that not every \(R \subseteq \mathcal{V}^0\) can be a redundant set. The problem whether there exists some \(\gamma\) such that \(R = \text{RD}(\gamma)\) for a given \(R\) is as hard as the branching bisimilarity checking problem [6].

**Main Result.** A process \(\alpha\) is a finite-state process if the reachable set \(\{\beta \mid \alpha \xrightarrow{\lambda_1} \alpha_1 \xrightarrow{\lambda_2} \ldots \xrightarrow{\lambda_k} \alpha_k = \beta \text{ and } k \in \mathbb{N}\}\) is finite. Given an equivalence relation \(\succeq\), we say that a BPA process \(\alpha\) is regular with respect to \(\succeq\), i.e. \(\succeq\)-REG, if \(\alpha \succeq \beta\) for some finite-state process \(\beta\). Note that \(\alpha\) and \(\beta\) can be defined in different systems.

In this paper we are interested in the equivalence checking and regularity checking problems with respect to \(\succeq\) on nBPA. They are defined as follows, assuming \(\succeq\) is an equivalence relation.

**Equivalence Checking with respect to \(\succeq\)**

- **Instance:** A nBPA system \((\mathcal{V}, \mathcal{A}, \Delta)\) and two processes \(\alpha\) and \(\beta\).
- **Question:** \(\alpha \succeq \beta\)?

**Regularity Checking with respect to \(\succeq\)**

- **Instance:** A nBPA system \((\mathcal{V}, \mathcal{A}, \Delta)\) and a process \(\alpha\).
- **Question:** \(\alpha \succeq\)-REG?
The following theorem states the two lower bounds proved in this paper.

\begin{itemize}
\item \textbf{Theorem 1.} On nBPA, for every equivalence \( \simeq \) such that \( \simeq \subseteq \approx \subseteq \approx \)
\item 1. equivalence checking with respect to \( \simeq \) is EXPTIME-hard; and
\item 2. regularity checking with respect to \( \simeq \) is PSPACE-hard.
\end{itemize}

## 3 Redundant Sets Construction

According to the branching bisimilarity checking algorithms of nBPA [4, 6], we know that the main exponential factor is exponentially many redundant sets. However there is no obvious evidence for the existence of such large number of redundant sets. Most nBPA systems in the literature have only polynomial ones. In this section, we design a scalable structure with exponentially many redundant sets. More specifically, we will define a nBPA system \( \Gamma(i) = (\mathcal{V}(i), \mathcal{A}(i), \Delta(i)) \) with parameter \( i \) such that (1) \( |\Gamma(i)| \) is a polynomial of \( i \); (2) and there are exactly \( 2^i \) redundant sets in \( \Gamma(i) \). We begin with \( \Gamma(2) \) as an example and explain its work mechanism in detail. Then we show how to scale \( \Gamma(2) \) to get \( \Gamma(i) \) for arbitrary \( i \) by incorporating the idea behind \( \Gamma(2) \). As an application of our approach, we design an \( n \)-bit binary counter that will be used in the sequel based on this structure.

### 3.1 A Small Example

The definition of \( \Gamma(2) = (\mathcal{V}(2), \mathcal{A}(2), \Delta(2)) \) is given as follows. \( \mathcal{V}(2) = \{ X, Y \} \uplus \{ A, B \} \uplus \{ A_B, B_A \} \), \( \mathcal{A}(2) = \{ a, b, d, \tau \} \) and the transition rules of \( \Delta(2) \) are shown in the following graph.

Every variable is normed and \( \mathcal{V}^0(2) = \{ X, Y \} \). To see why \( \Gamma(2) \) satisfies our requirement, we show that for each \( R \in \{ \emptyset, \{ X \}, \{ Y \}, \{ X, Y \} \} \), there is some \( \gamma \) such that \( \text{Rd}(\gamma) = R \). Clearly \( \text{Rd}(\emptyset) = \emptyset \). It is obvious that \( A \not\simeq A_B \) and \( B \not\simeq B_A \) due to the \( d \) action. Now we have \( XA \simeq A, YA \not\simeq A \) and \( XB \not\simeq B, YB \simeq B \), i.e., \( \text{Rd}(A) = \{ X \} \) and \( \text{Rd}(B) = \{ Y \} \). We claim that \( \text{Rd}(AB) = \text{Rd}(BA) = \{ X, Y \} \). Moreover, we have the following.

\begin{itemize}
\item \textbf{Claim.} \( \text{Rd}(A^nB) = \text{Rd}(B^nA) = \{ X, Y \} \) for all \( n \geq 1 \).
\end{itemize}

We first show that \( \text{Rd}(AB) = \{ X, Y \} \). It is clear \( X \in \text{Rd}(AB) \). It is not so obvious that \( Y \in \text{Rd}(AB) \). Let us consider the branching bisimulation game of \( YAB, AB \).

- If Attacker plays \( YAB \rightarrow AB \) or \( AB \xrightarrow{\lambda} \alpha \), then Defender plays an empty response or \( YAB \rightarrow AB \xrightarrow{\lambda} \alpha \) respectively. The next round configuration is an identical process pair. Defender wins afterward.
- If Attacker plays \( YAB \rightarrow AB \) then Defender responds with \( AB \rightarrow A_BB \). The game continues from \( (AB, A_BB) \).
- At the configuration \( (AB, A_BB) \) if Attacker plays an actions \( a \), then Defender responds with the same action and the configuration of next round is still \( (AB, A_BB) \). If Attacker plays an action \( b \), then after Defender’s response the game reaches the configuration \( (A_BB, A_BB) \). Defender wins. Attacker’s optimal choice is to play an action \( d \). Defender simply follows the suit and the game reaches the configuration \( (YB, B) \).
In a nutshell, from the configuration \((YAB, AB)\) Attacker and Defender’s optimal choices will lead the game to the configuration \((YB, B)\), i.e., \(YAB \simeq AB\) if and only if \(YB \simeq B\). Note that \(YB \simeq B\). It follows that \(YAB \simeq AB\) and \(Y \in \text{RD}(AB)\).

Now let us consider the general case \(\text{RD}(A^nB) = \{X, Y\}\) for all \(n \geq 1\). Clearly \(X \in \text{RD}(A^nB)\). We now show \(Y \in \text{RD}(A^nB)\). A key observation of the above game argument is that for each \(\gamma\) we have

\[
YA\gamma \simeq A\gamma \iff Y\gamma \simeq \gamma.
\]

Repeating (2) for \(n\) times we have \(YB \simeq B \implies YAB \simeq AB \implies \cdots \implies YA^nB \simeq A^nB\).

As a result \(Y \in \text{RD}(A^nB)\).

By a similar argument, we can show \(\text{RD}(B^nA) = \{X, Y\}\) for all \(n \geq 1\).

**Corollary 2.** For \(\gamma \in \{A, B\}^*\) we have (1) \(X \in \text{RD}(\gamma)\) iff \(A \in \text{VAR}(\gamma)\); (2) \(Y \in \text{RD}(\gamma)\) iff \(B \in \text{VAR}(\gamma)\).

### 3.2 Scalability

We now scale \(\Gamma(2)\) to get a nBPA system \(\Gamma(i) = (V(i), A(i), \Delta(i))\) with parameter \(i\) satisfying the following properties.

1. \(|\Gamma(i)|\) is a polynomial of \(i\).
2. \(|V(i)^0| = i\) and for each \(R \subseteq V(i)^0\) there is some \(\gamma \in (V(i) \setminus V(i)^0)^*\) such that \(\text{RD}(\gamma) = R\).

Note that the satisfaction of the second property would give rise to exactly \(2^i\) redundant sets in \(\Gamma(i)\). Before we give the definition of \(\Gamma(i)\), let us take a second look at \(\Gamma(2)\). We treat a configuration of the form \((X\gamma, \gamma)\) or \((Y\gamma, \gamma)\) as a redundancy test of \(X\) or \(Y\) on \(\gamma\).

Intuitively, the variable \(A\) has two roles. The first one is to pass the redundancy test of \(X\) as for each \(\gamma\) we have \(XA\gamma \simeq A\gamma\). The second one is to transfer the redundancy test of \(Y\) on \(A\gamma\) to the same test on \(\gamma\) as for each \(\gamma\) we have \(YA\gamma \simeq A\gamma\) if \(Y\gamma \simeq \gamma\). Similarly, \(B\) can pass redundancy test of \(Y\) and transfer the redundancy test of \(X\). Thus, we propose the following conditions to meet our requirement. For each \(Z \in V(i)^0\), there is some \(A \in V(i) \setminus V(i)^0\) so that for each \(\gamma\) it holds that:

- **(C1)** \(ZA\gamma \simeq A\gamma\), i.e., \(A\) passes the redundancy test of \(Z\); and
- **(C2)** for each \(Z' \in V(i)^0 \setminus \{Z\}\), \(Z'A\gamma \simeq A\gamma\) iff \(Z'\gamma \simeq \gamma\), i.e., \(A\) can transfer the redundancy test of each variable \(Z' \in V(i)^0 \setminus \{Z\}\).

**Remark.** In order to get a more flexible system to meet other requirements, the condition (C2) can be relaxed as “\(A\) transfers the redundancy tests of a portion of \(V(i)^0 \setminus \{Z\}\)”.

More specifically, "\(Z'A\gamma \simeq A\gamma\) iff \(Z'\gamma \simeq \gamma\)" holds for a *subset* of \(V(i)^0 \setminus \{Z\}\). We will see an example in Section 3.3.

Comparing \(\Gamma(2)\) with the conditions (C1) and (C2), we define \(V(i), A(i)\) and \(\Delta(i)\) as follows.

- \(V(i)^0 = \{Z_1, Z_2, \ldots, Z_i\}\);
- \(V(i) = V(i)^0 \cup \{A_1, A_2, \ldots, A_i\} \cup \{A_{j,k} \mid j \neq k, 1 \leq j, k \leq i\}\);
- \(A(i) = \{a_1, a_2, \ldots, a_i, d, \tau\}\);
- \(\Delta(i)\) contains the following rules, where \(1 \leq j, k, \ell \leq i, j \neq k, j \neq \ell\).

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Z_i \xrightarrow{a} \epsilon)</td>
<td>(Z_i \rightarrow \epsilon)</td>
</tr>
<tr>
<td>(A_j \xrightarrow{a} \epsilon)</td>
<td>(A_j \xrightarrow{a_j} A_j)</td>
</tr>
<tr>
<td>(A_{j,k} \xrightarrow{d} Z_k)</td>
<td>(A_{j,k} \xrightarrow{a_{j,k}} A_{j,k})</td>
</tr>
<tr>
<td>(A_{j,k} \xrightarrow{a_{j,k}} A_{j,k})</td>
<td>(A_{j,k} \xrightarrow{a_{j,k}} A_{j,k})</td>
</tr>
</tbody>
</table>

Clearly \(\Gamma(i)\) is of size \(O(i^2)\). The following justifies the conditions (C1) and (C2) in \(\Gamma(i)\).
Lemma 3. For each γ, 1 ≤ j, k ≤ i and j ≠ k it holds that
1. $Z_j A_j γ ≃ A_j γ$;
2. $Z_k A_j γ ≃ A_j γ$ iff $Z_k γ ≃ γ$.

With the help of Lemma 3, we can prove Proposition 4.

Proposition 4. $\text{Rd}(A_{j_1}, A_{j_2}, \ldots, A_{j_n}) = \bigcup_{k=1}^{n} \{Z_k\}$.

Proof. = If $k \in \bigcup_{i=1}^{n} \{j_i\}$, we show $Z_k \in \text{Rd}(A_{j_1}, A_{j_2}, \ldots, A_{j_n})$. Let $\ell'$ be the least number such that $j_{\ell'} = k$. By (1) of Lemma 3 we have $Z_k A_{j_{\ell'}} \ldots A_{j_n} ≃ A_{j_{\ell'}} \ldots A_{j_n}$. Now $Z_k A_{j_1} \ldots A_{j_n} ≃ A_{j_1} \ldots A_{j_n}$ can be derived by repeating (2) of Lemma 3 for $n - \ell'$ times.

= If $k \notin \bigcup_{i=1}^{n} \{j_i\}$, we show $Z_k \notin \text{Rd}(A_{j_1}, A_{j_2}, \ldots, A_{j_n})$. Note that $Z_k \not≃ \epsilon$. We are done by repeating (2) of Lemma 3 for $n$ times as $Z_k \not≃ \epsilon \implies Z_k A_{j_1} \not≃ A_{j_1} \implies \ldots \implies Z_k A_{j_1} A_{j_2} \ldots A_{j_n} \not≃ A_{j_1} A_{j_2} \ldots A_{j_n}$. ▶

3.3 An n-bit Binary Counter

An important application of the redundant sets construction is to implement an n-bit binary counter. The main idea is to use a redundant set of size $n$ to represent the value of an n-bit binary counter. The only challenging part is to define a proper structure of redundant sets so that it is fit to be manipulated by branching bisimulation games. Based on $\Gamma(2n)$, we implement an n-bit binary counter in the nBPA system $\Gamma_0 = (V_0, A_0, \Delta_0)$, where

$V_0 = V_0^0 \sqcup B \sqcup B'$,
$V_0^0 = \{Z_1^0, Z_1^1, Z_2^0, Z_2^1, \ldots, Z_n^0, Z_n^1\}$,
$B = \{B_i^0, B_i^1 | 1 ≤ i ≤ n\}$,
$B' = \{B_i^0(j, b') | (i \neq j) \land 1 ≤ i, j ≤ n \land b, b' \in \{0, 1\}\}$,
$A_0 = \{a_i^0, a_i^1 | 1 ≤ i ≤ n\} \sqcup \{d, \tau\}$.

And $\Delta_0$ contains the following rules, where $1 ≤ i, j, j' ≤ n$, $i \neq j$, $i \neq j'$ and $b, b', b'' \in \{0, 1\}$.

| $Z_i^0 \overset{a_i^0}{\rightarrow} \epsilon$ | $Z_i^0 \overset{a_i^1}{\rightarrow} \epsilon$ |
| $B_i^0 \overset{d}{\rightarrow} \epsilon$ | $B_i^0 \overset{a_i^b}{\rightarrow} B_i^b$ |
| $B_i^0(j, b') \overset{d}{\rightarrow} Z_i^b$ | $B_i^0(j, b') \overset{a_i^b}{\rightarrow} B_i^b(j, b')$ |
| $B_i^b \overset{a_i^{b'} \gamma}{\rightarrow} B_i^{b'} \gamma$ | $B_i^{b'}(j, b') \overset{a_i^{b''}}{\rightarrow} B_i^{b''}(j', b'')$ |

It is clear that $\text{Rd}(B_i^b) = \{Z_i^b\}$ for $1 ≤ i ≤ n$ and $b \in \{0, 1\}$. Intuitively, $B_i^b$ encodes the information that the $i$-th bit of the counter is $b$. To understand the structure of redundant sets in $\Gamma_0$, let us observe the three roles that $B_i^b$ plays in redundancy tests. Let $\gamma \in B^*$.

(P1) $Z_i^b B_i^b \gamma ≃ B_i^b \gamma$. This means that $B_i^b$ will pass the redundancy test of $Z_i^b$.

(P2) $Z_i^{b-i} B_i^b \gamma \not≃ B_i^{b-i} \gamma$. This means that $B_i^b$ will fail the redundancy test of $Z_i^{b-i}$.

(P3) For $j \neq i$ and $b', b'' \in \{0, 1\}$, $Z_j^b B_j^b \gamma \not≃ B_j^{b'} \gamma$ iff $Z_j^{b'} \gamma \not≃ \gamma$. This means $B_j^b$ will transfer the redundancy test of $Z_j^{b'}$ to next if $j \neq i$.

Note that the main structure of $\Gamma_0$ extends from $\Gamma(2n)$. The difference between $\Gamma_0$ and $\Gamma(2n)$ is that $\Gamma_0$ satisfies a relaxed version of the condition (C2). The properties of (P1) and (P2) together can be seen as a relaxed version of the condition (C2). The purpose of this design will be clear later. Using the above idea we have the following technical lemma.

Lemma 5. Suppose $\gamma \in B^*$, the following statements are valid.
1. \(Z^b_i \gamma \simeq \gamma \text{ iff there are } \gamma_1 \text{ and } \gamma_2 \text{ such that } \gamma = \gamma_1 B^b_i \gamma_2 \text{ and } B^1_i \not\in \text{VAR}(\gamma_1)\).
2. \(Z^b_i \gamma \simeq \gamma \text{ implies } Z^{1-b}_i \gamma \not\simeq \gamma\).

**Proof.** We only prove (1) here. (2) is a direct consequence of (1).

\((\Leftarrow)\) Let \(B^b_i \not\in \text{VAR}(\gamma_1)\). Clearly \(Z^b_i B^b_i \gamma \simeq B^b_i \gamma_2\). If \(\gamma_1 = \epsilon\) we are done; otherwise let \(\gamma_1 = B^b_i \ldots B^b_k\). Note that by assumption we have \(i_j \neq i\) for all \(1 \leq j \leq k\). We are done by repeating (P3) for \(k\) times from the equation \(Z^b_i B^b_i \gamma \simeq B^b_i \gamma_2\).

\((\Rightarrow)\) We prove it by contradiction. Suppose there is no \(\gamma_1 \) and \(\gamma_2\) such that \(\gamma = \gamma_1 B^b_i \gamma_2\) and \(B^1_i \not\in \text{VAR}(\gamma_1)\), we show \(Z^b_i \gamma \not\simeq \gamma\). There are two cases: (1) \(\gamma = \gamma_1 B^1_i \gamma_2\) and \(B^b_i, B^1_i \not\in \text{VAR}(\gamma_1)\); and (2) \(B^b_i, B^1_i \not\in \text{VAR}(\gamma)\). In the first case \(Z^b_i \gamma \not\simeq \gamma\) can be derived from \(Z^b_i B^1_i \gamma \not\simeq B^1_i \gamma_2\) by repeating (P3) for \(|\gamma_1|\) times; in the second case \(Z^b_i \gamma \not\simeq \gamma\) can be obtained from \(Z^b_i \not\simeq \epsilon\) by repeating (P3) for \(|\gamma|\) times.

**Definition 6.** A process \(\gamma \in \mathcal{B}^*\) is a valid encoding of an \(n\)-bit binary counter \(b_n b_{n-1} \ldots b_1\), notation \(\gamma \in \llbracket b_n b_{n-1} \ldots b_1 \rrbracket\), if for each \(1 \leq i \leq n\) there are \(\gamma_i \) such that \(\gamma = \gamma_1 B^b_i \gamma_i\) and \(B^1_i \not\in \text{VAR}(\gamma_i)\).

For a binary counter \(\gamma \in \llbracket b_n b_{n-1} \ldots b_1 \rrbracket\), we will use \(\sharp \gamma\) to denote the value \(\sum_{i=1}^n b_i \cdot 2^{i-1}\) in the sequel. One can see Definition 6 as the syntax of an \(n\)-bit binary counter in the system \(\Gamma_0\). This syntax allows us to update a “binary number” in an overwritten way. Suppose \(\gamma \in \llbracket b_n b_{n-1} \ldots b_1 \rrbracket\) and we want to flip the \(i\)-th “bit” of \(\gamma\) to get another “binary number” \(\gamma \). Then by Definition 6 we can simply let \(\sigma = B^1_i \gamma\), as one can verify \(\gamma \in \llbracket b_1 b_2 \ldots b_{i-1} b_{i} \rrbracket\) where \(b'_i = 1-b_i\) and \(b'_j = b_j\) for \(j \neq i\). By Lemma 5, we give the binary counter a semantic characterization in terms of redundant sets.

**Proposition 7.** Let \(\gamma\) be a process such that \(\gamma \in \mathcal{B}^*\), we have

\[
\gamma \in \llbracket b_n b_{n-1} \ldots b_1 \rrbracket \iff \text{RD}(\gamma) = \{Z^b_n, Z^{b_{n-1}}, \ldots, Z^b_1\).
\]

Proposition 7 provides us a way to test a specific “bit” with branching bisimulation games. Suppose \(\gamma \in \llbracket b_n b_{n-1} \ldots b_1 \rrbracket\) and we want to check whether \(b_i = b\). By Proposition 7, we only need to check if Defender has a w.s. in the branching bisimulation game \((\mathcal{Z^i}, \gamma, \gamma)\). The following lemma shows that we can also do bit test by weak bisimulation games.

**Lemma 8.** Suppose \(\gamma \in \mathcal{B}^*\), then \(Z^b_i \gamma \simeq \gamma \text{ iff } Z^b_i \gamma \approx \gamma\).

The following Lemma tells us how to test multiple bits. It is a simple consequence when applying Computation Lemma to Proposition 7 and Lemma 8.

**Lemma 9.** Let \(\gamma \in \llbracket b_n b_{n-1} \ldots b_1 \rrbracket\) and \(\alpha \in \{Z^1_1, Z^2_1, Z^3_1, \ldots, Z^0_n, Z^1_n\}^*\), then the following statements are valid.

1. \(\alpha \gamma \simeq \gamma \text{ iff } \alpha \in \{Z^1_1, Z^2_1, \ldots, Z^b_n\}^*\).
2. \(\alpha \gamma \approx \gamma \text{ iff } \alpha \in \{Z^1_1, Z^2_1, \ldots, Z^b_n\}^*\).

### 4 EXPTIME-hardness of Equivalence Checking

In this section, we show that branching bisimilarity on normed BPA is EXPTIME-hard by a reduction from Hit-or-Run game [10]. A Hit-or-Run game is a counter game defined by a tuple \(G = (S_0, S_1, s_0, s_1, \ldots, m_1)\), where \(S = S_0 \cup S_1\) is a finite set of states, \(s_0, s_1 \in S\) is the initial state, \(s_i \in S\) is the initial state, and \(m_1 \in S\) is the final value. We use \(s \xrightarrow{\ell} t\) to denote \((s, \ell, t) \in \rightarrow\) and require that \(\ell = 0\) or
\( \ell = 2^k \) for some \( k \). For each \( s \in S \) there is at least one rule \( (s, \ell, t) \in \to \). A configuration of \( \mathcal{G} \) is a pair \( (s, m) \in (S \cup \{s_i\}) \times \mathbb{N} \). The game is played by two players, named Player 0 and Player 1. Starting from the initial configuration \( (s_0, 0) \), the game \( \mathcal{G} \) proceeds in rounds according to the following rule: if the current configuration is \( (s, k) \in S \times \mathbb{N} \), then Player \( i \) chooses a rule of the form \( s \xrightarrow{\ell} t \) and the resulting new configuration is \( (t, k + \ell) \). If \( \mathcal{G} \) reaches \( (s_i, m) \) and \( m \neq m_i \) then Player 1 wins; if \( \mathcal{G} \) reaches the configuration \( (s_i, m) \) then Player 0 wins; if \( \mathcal{G} \) never reaches the final state \( s_i \), then Player 0 also wins. As a result, Player 0’s goal is to hit \( (s_i, m) \) or run from the final state \( s_i \). The problem of deciding the winner of Hit-or-Run game with all numbers represented in binary is EXPTIME-complete \([9, 10]\). The presented version of Hit-or-Run game is due to Kiefer \([10]\). Kiefer used it to establish the EXPTIME-hardness of strong bisimilarity on general BPA. The main technical result of the section is as follows.

**Proposition 10.** Given a Hit-or-Run game \( \mathcal{G} = (S_0, S_1, \to, s_0, s_i, m_i) \), a nBPA system \( \Gamma_1 = (\mathcal{V}_1, \mathcal{A}_1, \Delta_1) \) and two processes \( \xi, \xi' \in \mathcal{V}_1^* \) can be constructed in polynomial time such that

Player 0 has a w.s. in \( \mathcal{G} \iff \xi \simeq \xi' \iff \xi \approx \xi' \).

Our first lower bound (first item of Theorem 1) is a direct consequence of Proposition 10. Combining with the upper bound from \([6]\), we confirm the complete result.

**Corollary 11.** Branching bisimilarity checking on nBPA is EXPTIME-complete.

Now let us fix a Hit-or-Run game \( \mathcal{G} = (S_0, S_1, \to, s_0, s_i, m_i) \) for this section. Let \( \text{Op}(s) = \{\ell, t, \sigma \mid (s, \ell, t) \in \to\} \) and \( \text{Op} = \bigcup_{s \in S_0 \cup S_1} \text{Op}(s) \). We define the nBPA system \( \Gamma_1 = (\mathcal{V}_1, \mathcal{A}_1, \Delta_1) \) for Proposition 10 as follows.

\[
\begin{align*}
\mathcal{V}_1 &= V_0 \cup \mathcal{C} \cup \mathcal{F} \cup \mathcal{M}, \\
\mathcal{A}_1 &= A_0 \cup \{c, e, f, f', g\} \cup \{a(\ell, t) \mid (\ell, t) \in \text{Op}\}, \\
\Delta_1 &= \Delta_0 \cup \Delta_1'.
\end{align*}
\]

\( \Gamma_1 \) includes the n-bit counter system \( \Gamma_0 \) as a subsystem and use it represents the counter in the game \( \mathcal{G} \). We require that \( n \) and \( m_i \) satisfy the constrain \( n = \lceil \log_2 m_i \rceil + 1 \). This n-bit counter representation is sufficient for our purpose due to the following observation. When the counter value in \( \mathcal{G} \) is greater than \( 2^n - 1 \), Player 0 or Player 1’s object is to avoid or respectively to reach the final state \( s_i \) and the exact value of the counter no longer matters.

In the following we define the set \( \mathcal{C}, \mathcal{F} \) and \( \mathcal{M} \) and add rules to \( \Delta_1' \).

\( \text{(C).} \) The set \( \mathcal{C} \) is used to encode the control states of \( \mathcal{G} \) and is defined by

\[
\mathcal{C} = \{X(s), X'(s), Y(s), Y'(s) \mid s \in S \cup \{s_i\}\}.
\]

**Basic Idea.** Our reduction uses the branching (resp. weak) bisimulation game \( \mathcal{G}' \) starting from \( (\xi, \xi') \) to mimic the run of \( \mathcal{G} \) from \( (s_0, 0) \). We imagine that Defender simulates Player 0’s performance and Attacker simulates Player 1’s performance. For \( 0 \leq m < 2^n \), let Bin \( (m) \) be the unique n-bit binary representation of \( m \). The reduction will keep the following correspondence between \( \mathcal{G} \) and \( \mathcal{G}' \). If \( \mathcal{G} \) reaches a configuration \((s, m)\) with \( m < 2^n \), then \( \mathcal{G}' \) can reach a configuration \((X(s)\gamma, X'(s)\gamma)\) for some \( \gamma \in [\text{Bin}(m)] \) in a reasonable way; if \( \mathcal{G} \) reaches \((s, m)\) with \( m \geq 2^n \), then \( \mathcal{G}' \) can reach \((Y(s)\sigma, Y'(s)\sigma)\) for some \( \sigma \in [b_0 b_{n-1} \ldots b_1] \) in a reasonable way. Intuitively, \( Y(s) \) and \( Y'(s) \) indicate that the counter of \( \mathcal{G} \) overflows.
We do not track the exact value of the counter in that case. The two processes $\xi$ and $\xi'$ for Proposition 10 are defined by

$$
\xi = X(s_n)B_n^0B_{n-1}^0 \ldots B_1^0, \quad \xi' = X'(s_n)B_n^0B_{n-1}^0 \ldots B_1^0.
$$

(5)

Clearly $(\xi, \xi')$ corresponds to the initial configuration $(s_n, 0)$ in $G$.

$\mathcal{F}$. The set $\mathcal{F}$ is used to implement the Defender’s Forcing gadgets.

$$\mathcal{F} = \{ A(\ell, t), A'(\ell, t), E_s, F_s, E_s(\ell, t), F_s(\ell, t) \mid s \in S_0 \land (\ell, t) \in \mathrm{Op}(s) \}. $$

(6)

We add the following rules to $\Delta_1'$ for the variables in $C \cup \mathcal{F}$ to simulate the control flow of $G$.

1. Let $s \in S_1$. In $G$ Player 1 would choose one pair $(\ell, t)$ from $\mathrm{Op}(s)$, then correspondingly, rules (a1) and (a2) enable Attacker to choose the next move in $G'$.

   (a1). $X(s) \xrightarrow{a(\ell, t)} A(\ell, t)$, $X'(s) \xrightarrow{a'(\ell, t)} A'(\ell, t)$; $(\ell, t) \in \mathrm{Op}(s)$

   (a2). $Y(s) \xrightarrow{a(\ell, t)} Y(\ell, t)$, $Y'(s) \xrightarrow{a'(\ell, t)} Y'(\ell, t)$; $(\ell, t) \in \mathrm{Op}(s)$

2. Let $s \in S_0$. In $G$ Player 0 would choose one pair $(\ell, t)$ from $\mathrm{Op}(s)$. Rules (b1) (b2) and rules (b3) (b4) form two Defender’s Forcing gadgets [8], which allow Defender to choose the next move in $G'$. Let $(\ell, t), (\ell', t') \in \mathrm{Op}(s)$.

   (b1). $X(s) \xrightarrow{b(\ell, t)} E_s$, $X(s) \xrightarrow{b(\ell', t)} E_s(\ell, t)$, $X'(s) \xrightarrow{b(\ell, t)} E_s(\ell, t)$;

   (b2). $E_s \xrightarrow{a(\ell, t)} A(\ell, t)$, $E_s(\ell, t) \xrightarrow{a'(\ell, t)} A'(\ell, t)$, $E_s(\ell, t) \xrightarrow{a'(\ell', t')} A'(\ell', t'); (\ell', t') \neq (\ell, t)$

   (b3). $Y(s) \xrightarrow{b(\ell, t)} F_s$, $Y(s) \xrightarrow{b(\ell', t)} F_s(\ell, t)$, $Y'(s) \xrightarrow{b(\ell, t)} F_s(\ell, t)$;

   (b4). $F_s \xrightarrow{a(\ell, t)} Y(\ell, t)$, $F_s(\ell, t) \xrightarrow{a'(\ell, t)} Y'(\ell, t)$, $F_s(\ell, t) \xrightarrow{a'(\ell', t')} Y'(\ell'); (\ell', t') \neq (\ell, t)$

3. The following two rules for $X(s\ldots)$ and $X'(s\ldots)$ are used to test the value of counter with respect to $m_\gamma$. Let $\text{Bin}(m_\gamma) = b_0b_{n-1} \ldots b_1$.

   (c). $X(s) \xrightarrow{\gamma} Z_n^{b_n}Z_{n-1}^{b_{n-1}} \ldots Z_1^{b_1}$,

   $X'(s) \xrightarrow{\gamma} \varepsilon$.

   By Lemma 9, for $\gamma \in [b_nb_{n-1} \ldots b_1]$ we have $X(s_\gamma) \gamma \approx X'(s_\gamma) \gamma$ if $X(s_\gamma) \gamma \approx X'(s_\gamma) \gamma$ if $\gamma = m_\gamma$.

4. The following two rules are for $Y(s\ldots)$ and $Y'(s\ldots)$. Player 1 wins if $G$ reaches a configuration $(s\ldots, m)$ with $m \geq 2^n$. Correspondingly, $G'$ will reach a configuration $(Y(s\ldots)\sigma, Y'(s\ldots)\sigma)$ for some $\sigma \in [b_nb_{n-1} \ldots b_1]$. The following rules enable Attacker to win in this case by performing a special action that Defender cannot match.

   (d). $Y(s) \xrightarrow{\varepsilon}, Y'(s) \xrightarrow{\varepsilon}$.

$\mathcal{M}$. The set $\mathcal{M}$ is used to initiate the counter update operation and manipulate the n-bit binary counter.

$$\mathcal{M} = \{ \text{Add}(k, t), \text{Add}'(k, t), D(k, t), \text{D}(k, t), C(k, t, i), C'(k, t, i) \mid (\ell^k, t) \in \mathrm{Op} \land 0 \leq k < n \}. $$

(7)

The process pair $(A(\ell, t), A'(\ell, t))$ is used to implement the operation “increasing the counter by $\ell$ and goto state $t'$”. We add the following rules for this pair based on the value of $\ell$.

- $A(\ell, t) \xrightarrow{a} X(\ell, t)$ and $A'(\ell, t) \xrightarrow{a'} X'(\ell, t)$ if $\ell = 0$;
- $A(\ell, t) \xrightarrow{b} Y(\ell, t)$ and $A'(\ell, t) \xrightarrow{b'} Y'(\ell, t)$ if $\ell \geq 2^n$;
- $A(\ell, t) \xrightarrow{c} \text{Add}(\log \ell, t)$ and $A'(\ell, t) \xrightarrow{c'} \text{Add}'(\log \ell, t)$ if $0 < \ell < 2^n$.

If $\ell = 0$ or $\ell \geq 2^n$, the counter is either unchanged or overflow. In this case, we can directly switch the control state in $G'$. If $0 < \ell < 2^n$, $G'$ use the following mechanism to update the counter.
Binary Counter Manipulation. Suppose we have $\gamma \in \mathbb{B}^*$ representing a counter value, i.e. $\gamma \in [b_0,b_{n-1} \ldots b_1]$, and want to increase it by $2^k$, where $0 \leq k < n$. This operation has two possible outcomes. The counter is either updated to some $\sigma \in [b_0',b_{n-1} \ldots b_1']$ with $\exists \sigma = \gamma + 2^k$, or overflow if $\gamma + 2^k \geq 2^n$. Recall that $\bar{\gamma}$ and $\bar{\sigma}$ represent the values of $\gamma$ and $\sigma$. A key observation is that we can update $\gamma$ to $\sigma$ locally. Although there are $2^n$ many possible values for $\gamma$, we can write $\sigma$ as $\delta \gamma$ for exactly $n-k$ possible $\delta$. Indeed, let $\alpha(k,0)\ldots\alpha(k,n-k)$ and $\delta(k,0)\ldots\delta(k,1)\ldots\delta(k,n-k)$ be the processes defined by

$$
\begin{align*}
\alpha(k,0) &= Z^0_{k+1}, \\
\alpha(k,1) &= Z^1_{k+1}, \\
\vdots \\
\alpha(k,n-k-1) &= Z^0_nZ^1_{n-1} \ldots Z^1_{k+1}, \\
\alpha(k,n-k) &= Z^1_nZ^1_{n-1} \ldots Z^1_{k+1},
\end{align*}
$$

The set $\{ \gamma \mid \gamma \in [\text{Bin}(n)], 0 \leq m < 2^n \}$ can be divided into $n-k+1$ classes according to $\alpha(k,0), \alpha(k,1)\ldots\alpha(k,n-k)$. Intuitively, each $\alpha(k,i)$ encodes the bits which are flipped when increasing $\gamma$ by $2^k$. Each $\delta(k,i)$ encodes the corresponding effect of that operation. Let $i^*(k)$ be the maximal length of successive bits of $1$ starting from $b_{k+1}$ to $b_n$. Note that $i^*(k) = \Sigma_{j=0}^{n-k-1}(\Pi_{j=0}^{b_{k+1}+i})$. By Lemma 9, $\gamma \simeq \alpha(k,i)\gamma$ iff $i = i^*(k)$. If $i^*(k) < n-k$, then $\bar{\gamma} + 2^k < 2^n$. By the definition of $\delta(k,i^*(k))$ we can let $\sigma = \delta(k,i^*(k))\gamma$ and have $\exists \sigma = \bar{\gamma} + 2^k$. If $i^*(k) = n-k$, then $\bar{\gamma} + 2^k \geq 2^n$ and increasing $\gamma$ by $2^k$ will cause an overflow.

We now design a branching bisimulation game to simulate the addition operation based on the above idea. The following rules are for $M$, where $0 \leq i,j \leq n-k$.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A1)</td>
<td>$\text{Add}(k,t) \xrightarrow{\bar{\gamma}} D(k,t)$</td>
</tr>
<tr>
<td>(A2)</td>
<td>$\text{Add}(k,t) \xrightarrow{\epsilon} D(k,t,i)$</td>
</tr>
<tr>
<td>(A3)</td>
<td>$\text{D}(k,t) \xrightarrow{\epsilon} C(k,t,i)$</td>
</tr>
<tr>
<td>(A4)</td>
<td>$D(k,t,i) \xrightarrow{\epsilon} C^*(k,t,i)$</td>
</tr>
<tr>
<td>(A5)</td>
<td>$C(k,t,i) \xrightarrow{\epsilon} \alpha(k,i)$</td>
</tr>
<tr>
<td>(A6)</td>
<td>$C(k,t,i) \xrightarrow{\epsilon} X(t)\delta(k,i)$</td>
</tr>
<tr>
<td>(A7)</td>
<td>$C(k,t,n-k) \xrightarrow{\epsilon} Y(t)$</td>
</tr>
</tbody>
</table>

The correctness of the simulation is demonstrated by the following Lemma.

**Lemma 12.** Suppose $\gamma \in [b_0,b_{n-1} \ldots b_1]$ and $i^*(k) = \Sigma_{i=0}^{n-k-1}(\Pi_{j=0}^{b_{k+1}+i})$. In the branching bisimulation game starting from $(\text{Add}(k,t)\gamma, \text{Add}(k,t)\gamma)$ |

- **if** $\gamma = 2^k < 2^n$, then the optimal choices of Attacker and Defender will lead to the game reaching the configuration $(X(t)\delta(k,i^*(k))\gamma, X'(t)\delta(k,i^*(k))\gamma)$ with $\exists \delta(k,i^*(k))\gamma = \gamma + 2^k$, i.e., $\text{Add}(k,t)\gamma \simeq \text{Add}(k,t)\gamma$ iff $X(t)\delta(k,i^*(k))\gamma \simeq X'(t)\delta(k,i^*(k))\gamma$;

- **if** $\gamma + 2^k \geq 2^n$, then the optimal choices of Attacker and Defender will lead to the game reaching the configuration $(Y(t)\gamma, Y'(t)\gamma)$, i.e., $\text{Add}(k,t)\gamma \simeq \text{Add}(k,t)\gamma$ iff $Y(t)\gamma \simeq Y'(t)\gamma$.

**Proof.** Rules (A1) (A2) (A3) (A4) form a classical Defender’s Forcing gadget. Defender can use it to force the game from configuration $(\text{Add}(k,t)\gamma, \text{Add}(k,t)\gamma)$ to any configuration of the form $(C(k,t,i)\gamma, C^*(k,t,i)\gamma)$, where $0 \leq i \leq n-k$. Defender has to play carefully, as at the configuration $(C(k,t,i)\gamma, C^*(k,t,i)\gamma)$ Attacker can use rule (A5) to start up bits test by forcing the game to the configuration $(\alpha(k,i)\gamma, \gamma)$. By the definition of $\alpha(k,i)$ and Lemma 9, if $i = i^*(k)$ then Defender can survive the bits test as $\alpha(k,i^*(k))\gamma \simeq \gamma$; otherwise Defender will lose during the bits test as $\alpha(k,i)\gamma \not\simeq \gamma$ for $i \neq i^*(k)$. As a result Defender’s optimal
move is to force the configuration \((C(k, t, i^*(k))γ, C'(k, t, i^*(k))γ)\). In that case, Attacker’s optimal choice is to use rule (A6) or (A7) to increase the binary number \(γ\) by \(2^k\) or flag an overflow error. If \(i^*(k) < n - k\), the game reaches \((X(t)δ(k, i^*(k))γ, X'(t)δ(k, i^*(k))γ)\) by rule (A6). As \(δ(k, i^*(k))\) encodes the effect of bits change caused by increasing \(γ\) by \(2^k\), one can verify that \(\sharp(δ(k, i^*(k))γ) = 2γ + 2^k\). If \(i^*(k) = n - k\), the game goes to \((Y(t)γ, Y'(t)γ)\) by rule (A7).

 Remark. A process \(γ\) cannot perform an immediate internal action if there is no \(σ\) such that \(γ \rightarrow σ\). By construction \(X(t), X'(t), Y(t)\) and \(Y'(t)\) cannot perform immediate internal actions. As a result we can replace the branching bisimulation game of \(Add(k, t)γ, Add'(k, t)γ\) and “≈” in Lemma 12 with weak bisimulation game \(Add(k, t)γ, Add'(k, t)γ\) and “≃”.

Lemma 12 promises that the addition operation of the counter is implemented correctly in bisimulation games. We are ready to prove Proposition 10.

Proof of Proposition 10. Suppose \(G\) reaches \((s, m)\) for some \(s ∈ S_0 ⊔ S_1\) and \(m < 2^n\). The corresponding configuration of \(G'\) is \((X(s)γ, X'(s)γ)\) for some \(γ ∈ \{0, 1\}^m\). If \(s ∈ S_0\), then Player 0 chooses a rule \(s \rightarrow t\) and \(G\) proceeds to \((t, m + ℓ)\). We show how the branching bisimulation (resp. weak) bisimulation \(G'\) mimic this behavior while keep the correspondence between \(G\) and \(G'\). We only discuss the case \(s ∈ S_0\) here. The argument for \(s ∈ S_1\) is similar.

First by rules (b1) (b2), Defender forces to the configuration \((A(ℓ, t)γ, A'(ℓ, t)γ)\). If \(ℓ = 0\), then \(G'\) reaches \((X(ℓ)γ, X'(ℓ)γ)\). If \(ℓ ≥ 2^n\), then \(G'\) reaches \((Y(t)γ, Y'(t)γ)\). If \(0 < ℓ < 2^n\), then \(G'\) first reaches \((Add(\log ℓ, t)γ, Add'(\log ℓ, t)γ))\). Now the binary counter in \(G'\) will be updated according to \(ℓ\). By Lemma 12, if \(m + ℓ < 2^n\), then the optimal play of Attacker and Defender will lead to \((X(t)σ, X'(t)σ)\) with \(2σ = 2γ + ℓ\). If \(m + ℓ ≥ 2^n\), then the optimal configuration for both Attacker and Defender is \((Y(t)γ, Y'(t)γ)\).

Once \(G\) reaches a configuration \((s', m)\) with \(m ≥ 2^n\) and \(s' ≠ s_+\), \(G'\) reaches \((Y(s')σ, Y'(s')σ)\) for some \(σ\). By rules (a2) (b3) (b4), \(G'\) will only keep track of the state shift of \(G\) afterward.

If Player 0 has a strategy to hit \((s_+, m_+)\) or run from \(s_+\) then Defender can mimic the strategy to push \(G'\) from \((ξ, ξ')\) to a configuration \((X(s_+)γ, X'(s_+)γ)\) for some \(γ ∈ \{0, 1\}^{m_+}\) or force \(G'\) to be played infinitely. By rule (c) and Lemma 9, \(X(s_+)γ ≃ X'(s_+)γ\). It follows that \(ξ ≃ ξ'\). If Player 1 has a strategy such that no matter how Player 0 chooses, the game will hit some configuration of the form \((s_+, m)\) with \(m ≠ m_+\). Then Attacker can mimic the strategy to force \(G'\) from \((ξ, ξ')\) to \((X(s_+)γ, X'(s_+)γ)\) for some \(γ ∈ \{0, 1\}^m\) if \(m < 2^n\), or to \((Y(s_+)σ, Y'(s_+)σ)\) for some \(σ\) if \(m ≥ 2^n\). By rule (c) and Lemma 9, \(X(s_+)γ ≠ X'(s_+)γ\). By rule (d), \(Y(s_+)σ ≠ Y'(s_+)σ\). It follows that \(ξ ≃ ξ'\).

5 PSPACE-hardness of Regularity Checking

Srba [17] proved that weak bisimilarity can be reduced to weak regularity under a certain condition. We can verify that his original construction also works for branching regularity.

Theorem 13 (Srba[17]). Given a BPA system \(Γ\) and two process \(α\) and \(β\), one can construct in polynomial time a new BPA system \(Γ'\) and a process \(γ\) such that (1) \(γ\) is \(≈\)-REG iff \(α ≃ β\) and both \(α\) and \(β\) are \(≈\)-REG; (2) \(γ\) is \(≃\)-REG iff \(α ≃ β\) and both \(α\) and \(β\) are \(≃\)-REG; and (3) \(γ\) is normed iff \(α\) and \(β\) are normed.

By Theorem 13, to get a lower bound of branching regularity on normed BPA we only need to prove a lower bound of branching bisimilarity. Note that we cannot adapt the previous reduction to get an EXPTIME-hardness result for regularity as \(ξ\) and \(ξ'\) for Proposition 10.
are neither ≼-REG nor ≼-REG. Srba proved that weak bisimilarity is PSPACE-hard [14] and the two processes for the construction are ≼-REG. This implies that weak regularity of normed BPA is PSPACE-hard. However, the construction in [14] does not work for branching bisimilarity. We can fix this problem by adapting the previous redundant sets construction.

**Proposition 14.** Given a QSAT formula  \( \mathfrak{F} \), we can construct a normed BPA system \( \Gamma_2 = (V_2, A_2, \Delta_2) \) and two normed processes \( X_1 \) and \( X_1' \) satisfying the following conditions.

1. If \( \mathfrak{F} \) is true then \( X_1 \simeq X_1' \).
2. If \( \mathfrak{F} \) is false then \( X_1 \not\simeq X_2 \).
3. \( \alpha \) and \( \beta \) are both \( \simeq \)-REG.

Combining Theorem 13 and Proposition 14 we get our second lower bound result (second item of Theorem 1).

Now let us first fix a QSAT formula

\[
\mathfrak{F} = \forall x_1 \exists y_1 \forall x_2 \exists y_2 \ldots \forall x_m \exists y_m . (C_1 \land C_2 \land \cdots \land C_n)
\]

where \( C_1 \land C_2 \land \cdots \land C_n \) is a conjunctive normal form with boolean variables \( x_1, x_2, \ldots, x_m \) and \( y_1, y_2, \ldots, y_m \). Consider the following game interpretation of the QSAT formula \( \mathfrak{F} \).

There are two players, \( X \) and \( Y \), who are trying to give an assignment in rounds to all the variables \( x_1, y_1, x_2, y_2, \ldots, x_m, y_m \). At the \( i \)-th round, player \( X \) first assigns a boolean value \( b_i \) to \( x_i \) and then \( Y \) assigns a boolean value \( b'_i \) to \( y_i \). After \( m \) rounds we get an assignment \( A = \bigcup_{i=1}^{m} \{ x_i \mapsto b_i, y_i \mapsto b'_i \} \). If \( A \) satisfies \( C_1 \land C_2 \land \cdots \land C_n \), then \( Y \) wins; otherwise \( X \) wins.

It is easy to see that \( \mathfrak{F} \) is true iff \( Y \) has a winning strategy. This basic idea of constructing \( \Gamma_2 \) is to design a branching (resp. weak) bisimulation game to mimic the QSAT game on \( \mathfrak{F} \). This method resembles the ideas in the previous works [14, 15, 16]. The substantial new ingredient in our construction is \( \Gamma(n) \), introduced in Section 3. \( \Gamma_2 \) contains \( \Gamma(n) \) as a subsystem and uses it to encode partial assignments in the QSAT game. For \( i \in \{1, 2, \ldots, m\} \) and \( b \in \{0, 1\} \), let \( \alpha(i, b) \) and \( \beta(i, b) \) be the processes defined as follows.

- \( \alpha(i, b) = A_{i_1} A_{i_2} \ldots A_{i_k} \). If \( b = 1 \), then \( 1 \leq i_1 < i_2 < \cdots < i_k \leq n \) are all the indices of clauses in \( \mathfrak{F} \) that \( x_i \) occurs; if \( b = 0 \), then \( 1 \leq i_1 < i_2 < \cdots < i_k \leq n \) are all the indices of clauses that \( \overline{x_i} \) occurs.

- \( \beta(i, b) = A_{i_1} A_{i_2} \ldots A_{i_k} \). If \( b = 1 \), then \( 1 \leq i_1 < i_2 < \cdots < i_k \leq n \) are all the indices of clauses in \( \mathfrak{F} \) that \( y_i \) occurs; if \( b = 0 \), then \( 1 \leq i_1 < i_2 < \cdots < i_k \leq n \) are all the indices of clauses that \( \overline{y_i} \) occurs.

An assignment \( A = \bigcup_{i=1}^{m} \{ x_i \mapsto b_i, y_i \mapsto b'_i \} \) is represented by the process \( \gamma(A) \) defined by

\[
\gamma(A) = \beta(m, b'_m) \alpha(m, b_m) \ldots \beta(1, b'_1) \alpha(1, b_1).
\]

Clearly \( A \) satisfies \( C_1 \land C_2 \land \cdots \land C_n \) iff \( \text{VAR}(\gamma(A)) = \{A_1, A_2, \ldots, A_n\} \). The following lemma tells us how to test the satisfiability of \( A \) by bisimulation games.

**Lemma 15.** Suppose \( \gamma \in \{A_1, A_2, \ldots, A_n\}^* \). The following statements are equivalent.

1. \( Z_1 Z_2 \ldots Z_n \gamma \simeq \gamma \).
2. \( Z_1 Z_2 \ldots Z_n \gamma \simeq \gamma \).
3. \( \text{VAR}(\gamma) = \{A_1, A_2, \ldots, A_n\} \).

Now the normed BPA system \( \Gamma_2 = (V_2, A_2, \Delta_2) \) for Proposition 14 is defined by

\[
V_2 = V(n) \cup \{ X_i, Y_i(1), Y_i(2), Y_i(3) \mid 1 \leq i \leq m \} \cup \{ X_{m+1}, X'_{m+1} \},
\]

\[
A_2 = A(n) \cup \{ c_0, c_1, e \},
\]

\[
\Delta_2 = \Delta(n) \cup \Delta'.
\]

And \( \Delta' \) contains the following rules, where \( 1 \leq i \leq m \).
20:14 Two Lower Bounds for BPA

Proof Sketch of Proposition 14. It is clear that both $X_1$ and $X_1'$ are $\approx$-REG. Now consider the branching (resp. weak) bisimulation game starting from $(X_1, X_1')$. One round of QSAT game on $\mathfrak{g}$ will be simulated by three rounds of branching (resp. weak) bisimulation games. Suppose player $\mathcal{X}$ assigns $b_i$ to $x_i$ and then player $\mathcal{Y}$ assigns $b'_i$ to $y_i$ in the $i$-th round of QSAT game. Then in the branching (resp. weak) bisimulation game, Attacker uses rule (1) and (2) to push $\alpha(i, b)$ to the stack in one round; then in the following two rounds, by Defender’s Forcing (rule (3) (4) (5) (6) (7)), Defender pushes $\beta(i, b'_i)$ to the stack. In this way, the branching (resp. weak) bisimulation game reaches a configuration in the form $(X_{m+1}\gamma(A), X'_{m+1}\gamma(A))$ after $3m$ rounds. Here $A = \bigcup_{i=1}^{m} \{ x_i \mapsto b_i, y_i \mapsto b'_i \}$ is an assignment that $\mathcal{X}$ and $\mathcal{Y}$ generates. It follows from Lemma 15 and rule (8) that if $\mathcal{Y}$ has a w.s. then Defender can ensure that $X_{m+1}\gamma(A) \ncong X'_{m+1}\gamma(A)$ by the strategy; otherwise if $\mathcal{X}$ has a w.s. then Attacker can use it to reach a pair that guarantee $X_{m+1}\gamma(A) \ncong X'_{m+1}\gamma(A)$.

6 Conclusion

The initial motivation of this paper is to confirm the EXPTIME-hardness claim of branching bisimilarity of nBPA [5, 6]. The main contribution of this work is a technique to design a structure with a large number of redundant sets. One can use the binary counter constructed in Section 3.3 to represent the content of the tape of an ALBA. The construction allows to overwrite symbols in cells of the tape and to check their values. This would produce another EXPTIME-hard reduction for the branching bisimilarity of nBPA from the acceptance problem of ALBA. The PSPACE-hard lower bound of branching regularity is a byproduct in the development of this work. There is a gap in the complexity of branching regularity of nBPA. Czerwiński and Jančar proved that it can be decided in NEXPTIME [4]. An EXPTIME algorithm is also feasible based on the exponentially large bisimulation base [6]. Whether there exists a PSPACE algorithm is a natural further question. Another interesting research direction concerns the branching bisimilarity of nBPP (normed Basic Parallel Process), the parallel counterpart of nBPA in the PRS hierarchy [12]. Czerwiński, Hofman and Lasota proved the decidability of this problem [3]. However, the complexity is not clear. The current lower bound is PSPACE-hard [17]. For the branching regularity of nBPP, the lower bound is also PSPACE-hard [17], while the decidability is open. There are huge gaps worth further study. We hope that the technique introduced in this paper could shed some new light on these problems.

Acknowledgments. We thank Prof. Petr Jančar for his kind reminder of the EXPTIME-hardness claim and the members of BASICS for their interest. We thank Prof. Yuxi Fu, Dr. Huan Long and the anonymous reviewers for their comments to improve this paper.
References


Infinite-Duration Bidding Games

Guy Avni\textsuperscript{1}, Thomas A. Henzinger\textsuperscript{2}, and Ventsislav Chonev\textsuperscript{3}

\textsuperscript{1} IST Austria, Klosterneuburg, Austria
\textsuperscript{2} IST Austria, Klosterneuburg, Austria
\textsuperscript{3} Max Planck Institute for Software Systems (MPI-SWS), Saarbrücken, Germany

Abstract

Two-player games on graphs are widely studied in formal methods as they model the interaction between a system and its environment. The game is played by moving a token throughout a graph to produce an infinite path. There are several common modes to determine how the players move the token through the graph; e.g., in turn-based games the players alternate turns in moving the token. We study the bidding mode of moving the token, which, to the best of our knowledge, has never been studied in infinite-duration games. Both players have separate budgets, which sum up to 1. In each turn, a bidding takes place. Both players submit bids simultaneously, and a bid is legal if it does not exceed the available budget. The winner of the bidding pays his bid to the other player and moves the token. For reachability objectives, repeated bidding games have been studied and are called Richman games \cite{richman1, richman2}. There, a central question is the existence and computation of threshold budgets; namely, a value $t \in [0,1]$ such that if Player 1’s budget exceeds $t$, he can win the game, and if Player 2’s budget exceeds $1 - t$, he can win the game. We focus on parity games and mean-payoff games. We show the existence of threshold budgets in these games, and reduce the problem of finding them to Richman games. We also determine the strategy-complexity of an optimal strategy. Our most interesting result shows that memoryless strategies suffice for mean-payoff bidding games.

1998 ACM Subject Classification J.4 Social and Behavioral Sciences, F.1.2 Modes of Computation

Keywords and phrases Bidding games, Richman games, Parity games, Mean-payoff games

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2017.21

1 Introduction

Two-player infinite-duration games on graphs are an important class of games as they model the interaction of a system and its environment. Questions about automatic synthesis of a reactive system from its specification \cite{synthesis} are reduced to finding a winning strategy for the “system” player in a two-player game. The game is played by placing a token on a vertex in the graph and allowing the players to move it throughout the graph, thus producing an infinite trace. The winner or value of the game is determined according to the trace. There are several common modes to determine how the players move the token that are used to model different types of systems (c.f., \cite{modes}). The most well-studied mode is turn-based, where the vertices are partitioned between the players and the player who controls the vertex on which the token is placed, moves it. Other modes include probabilistic and concurrent moves.

* This research was supported in part by the Austrian Science Fund (FWF) under grants S11402-N23 (RiSE/SHiNE) and Z211-N23 (Wittgenstein Award).

© Guy Avni, Thomas A. Henzinger, and Ventsislav Chonev; licensed under Creative Commons License CC-BY

28th International Conference on Concurrency Theory (CONCUR 2017).
Editors: Roland Meyer and Uwe Nestmann; Article No. 21; pp. 21:1–21:18
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany
We study a different mode of moving, which we refer to as bidding, and to the best of our knowledge, has never been studied for infinite-duration games. Both players have budgets, where for convenience, we have $B_1 + B_2 = 1$. In each turn a bidding takes place for the right to move the token. The players submit bids simultaneously, where a bid is legal if it does not exceed the available budget. Thus, a bid is a real number in $[0, B_i]$, for $i \in \{1, 2\}$. The player who bids higher pays the other player, and decides where the token moves. Draws can occur and one needs to devise a mechanism for resolving them (e.g., giving advantage to Player 1), and our results do not depend on a specific mechanism.

Bidding arises in many settings and we list several examples below. The players in a two-player game often model concurrent processes. Bidding for moving can model an interaction with a scheduler. The process that wins the bidding gets scheduled and proceeds with its computation. Thus, moving has a cost and processes are interested in moving only when it is critical. When and how much to bid can be seen as quantifying the resources that are needed for a system to achieve its objective, which is an interesting question. Other takes on this problem include reasoning about which input signals need to be read by the system at its different states [20, 2] as well as allowing the system to read chunks of input signals before producing an output signal [28, 27, 33]. Also, our bidding game can model scrip systems that use internal currencies for bidding in order to prevent “free riding” [31]. Such systems are successfully used in various settings such as databases [43], group decision making [42], resource allocation, and peer-to-peer networks (see [29] and references therein). Finally, repeated bidding is a form of a sequential auction [37], which is used in many settings including online advertising.

Recall that the winner or value of the game is determined according to the outcome, which is an infinite trace. There are several well-studied objectives in games. The simplest objective is reachability, where Player 1 has a target vertex and a trace is winning for him iff it visits the target. Bidding reachability games are equivalent to Richman games [36, 35], named after David Richman. Richman games are the first to study the bidding mode of moving. The central question that is studied on Richman games regards a threshold budget, which is a function $\text{Thresh} : V \to [0, 1]$ such that if Player 1’s budget exceeds $\text{Thresh}(v)$ at a vertex $v$, then he has a strategy to win the game. On the other hand, if Player 2’s budget exceeds $1 - \text{Thresh}(v)$, he can win the game (recall that the budgets add up to 1).

In [36, 35], the authors show that threshold budgets exist, are unique, and that finding them is in NP. We slightly improve their result by showing that the problem is in NP and coNP. We illustrate the bidding model and the threshold problem in the following example.

**Example 1.** Consider for example, the bidding reachability game that is depicted in Figure 1. Player 1’s goal is to reach $t$, and Player 2’s goal is to prevent this from happening. How much budget suffices for Player 1 to guarantee winning? Clearly, even if Player 1 has all the budget, he cannot win in $v_1$, thus $\text{Thresh}(v_1) = 1$. Similarly, even if Player 2 has all the budget in $t$, Player 1 has already won, thus $\text{Thresh}(t) = 0$. We show a naive solution in which Player 1 wins when his budget exceeds 0.75. Indeed, if Player 1’s budget is $0.75 + \epsilon$, for $\epsilon > 0$, then since the budgets add up to 1, Player 2’s budget is $0.25 - \epsilon$. In the first turn, Player 1 bids $0.25 + \frac{\epsilon}{2}$ and wins the bidding since Player 2 cannot bid above 0.25. He pays his bid to Player 2 and moves the token to $v_2$. Thus, at the end of the round, the budgets are $0.5 + \frac{\epsilon}{2}$ and $0.5 - \frac{\epsilon}{2}$ and the token is on $v_2$. In the second bidding, Player 1 bids all his budget, wins the bidding since Player 2 cannot bid above 0.5, moves the token to $t$, and wins the game. It turns out that the threshold budgets are lower: it follows from Theorem 3 that they are $\text{Thresh}(v_0) = 2/3$ and $\text{Thresh}(v_2) = 1/3$. ◀
We introduce and study infinite duration bidding games with richer qualitative objectives as well as quantitative objectives. Parity games are an important class of qualitative games as the problem of reactive synthesis from LTL specifications is reduced to a parity game. The vertices in a parity game are labeled by an index in \( \{0, \ldots, d\} \), for some \( d \in \mathbb{N} \), and an infinite trace is winning for Player 1 iff the parity of the maximal index that is visited infinitely often is odd. The quantitative games we focus on are mean-payoff games. An infinite outcome has a value, which can be thought of as the amount of money that Player 1 pays Player 2. Accordingly, we refer to the players in a mean-payoff game as Maximizer (Max, for short) and Minimizer (Min, for short). The vertices of a mean-payoff game are labeled by values in \( \mathbb{Z} \). Consider an infinite trace \( \pi \). The energy of a prefix \( \pi^n \) of length \( n \) of \( \pi \), denoted \( E(\pi^n) \), is the sum of the values it traverses. The mean-payoff value of \( \pi \) is \( \lim_{n \to \infty} \frac{E(\pi^n)}{n} \). We are interested in cases where Min can guarantee a non-positive mean-payoff value. It suffices to show that he can guarantee that an infinite outcome \( \pi \) either has infinitely many prefixes with \( E(\pi^n) = 0 \), or that the energy is bounded, thus there is \( N \in \mathbb{N} \) such that for every \( n \in \mathbb{N} \), we have \( E(\pi^n) \leq N \). We stress the point that there are two “currencies” in the game: a “monopoly money” that is used to determine who moves the token and which the players do not care about once the game ends, and the values on the vertices, which is the value that Min and Max seek to minimize and maximize, respectively. We illustrate mean-payoff games with the following example.

**Example 2.** Consider the mean-payoff bidding game that is depicted in Figure 1, where for convenience the values are placed on the edges and not on the vertices. We claim that Min has a strategy that guarantees a non-positive mean-payoff value. Without loss of generality, Max always chooses the 1-valued edge. Min’s strategy is a tit-for-tat-like strategy, and he always takes the \((-1)\)-valued edge. The difficulty is in finding the right bids. Initially, Min bids 0. Assume Max wins a bidding with \( b > 0 \). Min will try and match this win: he bids \( b \) until he wins with it. Let \( b_1, \ldots, b_n \) be Max’s winning bids before Min wins with \( b \). We call these un-matched bids. The next bid Min attempts to match is \( b' = \min_{1 \leq i \leq n} b_i \); he bids \( b' \) until he wins with it, and continues similarly until all bids are matched.

We claim that the tit-for-tat strategy guarantees a non-positive mean-payoff value. Observe first that if a prefix of the outcome has \( k \) unmatched bids, then the energy is \( k \). In particular, if all bids are matched, the energy is 0. Suppose Min bids \( b \). We claim that the number of un-matched bids is at most \( \lceil 1/b \rceil \). Otherwise, since \( b \) is less than all other un-matched bids, Max would need to invest more than a budget of 1. It follows that an infinite outcome that never reaches energy level 0 has bounded energy, thus the mean-payoff value is non-positive.

We study the existence and computation of threshold budgets in parity and mean-payoff bidding games. Also, we determine the strategy complexity that is necessary for winning. Recall that a winning strategy in a game typically corresponds to an implementation of a system. A strategy that uses an unbounded memory, like the tit-for-tat strategy above, is not useful for implementing. Thus, our goal is to find strategies that use little or no memory, which are known as memoryless strategies.
We show that parity bidding games are linearly-reducible to Richman games allowing us to obtain all the positive results from these games; threshold budgets exist, are unique, and computing them is no harder than for Richman games, i.e., the problem is in NP and coNP. We find this result quite surprising since for most other modes of moving, parity games are considerably harder than reachability games. The crux of the proof considers bottom strongly-connected components (BSCCs, for short) in the arena, i.e., SCCs with no exiting edges. We show that in a strongly connected bidding parity game, exactly one of the players wins with every initial budget, thus the threshold budgets of the vertices of a BSCC are in \( \{0, 1\} \). If the vertex with highest parity in a BSCC is odd, then Player 1 wins, i.e., the threshold budgets are all 0, and otherwise Player 2 wins, i.e., the threshold budgets are all 1. We can thus construct a Richman game by setting the target of Player 1 to the BSCCs that are winning for him and the target of Player 2 to the ones that are winning for him. Moreover, we show that memoryless strategies are sufficient for winning in these games.

We proceed to study mean-payoff bidding games. We adapt the definition of threshold values; we say that \( t \in [0, 1] \) is a threshold value for Min if with a budget that exceeds \( t \), Min can guarantee a non-positive mean-payoff value. On the other hand, if Max's budget exceeds \( 1 - t \), he can guarantee a positive mean-payoff value. We show that threshold values exist and are unique in mean-payoff bidding games. The crux of the existence proof again considers the BSCCs of the game. We show that in a strongly-connected mean-payoff bidding game, the threshold budgets are in \( \{0, 1\} \), thus again either Min “wins” or Max “wins” the game. Moreover, this classification can be determined in NP and coNP, thus the complexity of solving bidding mean-payoff games coincides with Richman games. Our results for strongly-connected games are obtained by developing the connection that was observed in [36, 35] between the threshold budget and the reachability probability in a probabilistic model on the same structure as the game. We show a connection between bidding mean-payoff games and one-counter 2.5-player games [14, 13] to prove the classification of BSCCs. In turn, these games are equivalent to discrete quasi-birth-death processes [24] and generalize solvency games [11], which can be thought of as a rewarded Markov decision process with a single vertex.

The classification above is existential in nature and does not provide any insight on how a player guarantees a mean-payoff value. Our most technically challenging results concern the constructions strategies for Min and Max. The challenging part of the construction is reasoning about strongly-connected bidding mean-payoff games. Consider a strongly-connected game in which Min can guarantee a non-positive mean-payoff value. The idea of our construction is to tie between changes in Min’s budget with changes in the energy; investing one unit of budget (with the appropriate normalization) implies a decrease of a unit of energy, and on the other hand, an increase of a unit of energy implies a gain of one unit of budget. Since the budgets are bounded by 1, the value cannot increase arbitrarily. Finding the right bids in a general SCC is not trivial, and we find our solution to be surprisingly elegant. The case where Max can guarantee a positive mean-payoff value, is more challenging. Unlike a memoryless strategy for Min, the normalization factor must decrease as the value increases so that Max does not exhaust his budget. We show constant memory strategies in general and identify a fragment in which we show memoryless strategies.

Further bidding games

Variants of bidding games where studied in the past. Already in [35] several variants are studied including a poorman version in which the winner of the bidding pays the bank, thus the amount of money in the game decreases as the game proceeds. Motivated by recreational
games, e.g., bidding chess, discrete bidding games are studied in [23], where the money is divided into chips, so a bid cannot be arbitrarily small as in the bidding games we study. In all-pay bidding games [38], the players all pay their bids to the bank. Non-zero-sum two-player games were recently studied in [30]. They consider a bidding game on a directed acyclic graph. Moving the token throughout the graph is done by means of bidding. The game ends once the token reaches a sink, and each sink is labeled with a pair of payoffs for the two players that do not necessarily sum up to 0. They show existence of subgame perfect equilibrium for every initial budget and a polynomial algorithm to compute it.

Due to lack of space, most of the proofs appear in the full version [7].

2 Preliminaries

An arena is a pair \( \langle G, \alpha \rangle \), where \( G \) is a directed graph and \( \alpha \) is an objective. A game is played on an arena as follows. A token is placed on a vertex in the arena and the players move it throughout the graph. The outcome is an infinite path \( \pi \). The winner or value is determined according to \( \pi \) and \( \alpha \) as we elaborate below. There are several common modes in which the players move the token. In turn-based games the vertices are partitioned between the players and the player who controls the vertex on which the token is placed, moves it. Another mode is probabilistic choices, where the game can be thought of as a Markov chain, thus the edges are labeled with probabilities, and the edge on which the token proceeds is chosen randomly. A combination of these two modes is called 2.5-player games, where the vertices are partitioned into three sets: Player 1 vertices, Player 2 vertices, and probabilistic vertices. Finally, in concurrent games, each player has a possible (typically finite) set of actions he can choose from in a vertex. The players select an action simultaneously, and the choice of actions dictates to which vertex the token moves.

We study a different mode of moving, which we call bidding. Both players have budgets, where for convenience, we have \( B_1 + B_2 = 1 \). In each turn, a bidding takes place to determine who moves the token. Both players submit bids simultaneously, where a bid is a real number in \( [0, B_i] \), for \( i \in \{1, 2\} \). The player who bids higher pays the other player and decides where the token moves. Note that the sum of budgets always remains 1. While draws can occur, in the questions we study we try avoid the issue of draws.

A strategy prescribes to a player which action to take in a game, given a finite history of the game, where we define these two notions below. In 2.5-player games, histories are paths and actions are vertices. Thus, a strategy for Player \( i \), for \( i \in \{1, 2\} \), takes a finite path that ends in a Player \( i \) vertex, and prescribes to which vertex the token moves to next. In bidding games, histories and strategies are more complicated as they maintain the information about the bids and winners of the bids. A history is a sequence of the form \( v_0, \langle v_1, b_1, i_1 \rangle, \langle v_2, b_2, i_2 \rangle, \ldots, \langle v_k, b_k, i_k \rangle \in V \cdot (V \times [0, 1] \times \{1, 2\})^* \), where, for \( j \geq 1 \), in the \( j \)-th round, the token is placed on vertex \( v_{j-1} \), the winning bid is \( b_j \), and the winner is Player \( i_j \), and Player \( i_j \) moves the token to vertex \( v_j \). An action for a player is \( \langle b, v \rangle \in ([0, 1] \times V) \), where \( b \) is the bid and \( v \) is the vertex to move to upon winning. An initial vertex \( v_0 \) and strategies \( f_1 \) and \( f_2 \) for Players 1 and 2, respectively, determine a unique outcome \( \pi \) for the game, denoted \( \text{out}(v_0, f_1, f_2) \), which is an infinite sequence in \( V \cdot (V \times [0, 1] \times \{1, 2\})^\omega \). We sometimes abuse notation and refer to \( \text{out}(v_0, f_1, f_2) \) as a finite prefix of the infinite outcome.

We drop \( v_0 \) when it is clear from the context. We define the outcome inductively. The first element of the outcome is \( v_0 \). Suppose \( \pi_1, \ldots, \pi_j \) is defined. The players bids are given by \( \langle b_1, v_1 \rangle = f_1(\pi_1, \ldots, \pi_j) \) and \( \langle b_2, v_2 \rangle = f_2(\pi_1, \ldots, \pi_j) \). If \( b_1 > b_2 \), then \( \pi_{j+1} = \langle v_1, b_1, 1 \rangle \), and dually when \( b_1 < b_2 \), we have \( \pi_{j+1} = \langle v_2, b_2, 2 \rangle \). We assume there is some tie-breaking
mechanism that determines who the winner is when \( b_1 = b_2 \), and our results are not affected by what the tie-breaking mechanism is. Consider a finite outcome \( \pi \). The payment of Player 1 in \( \pi \), denoted \( B_1(\pi) \), is \( \sum_{i \leq j < |\mathcal{S}|} (-1)^{j-i} b_j \), and Player 2’s payment, denoted \( B_2(\pi) \) is defined similarly. For \( i \in \{1, 2\} \), consider an initial budget \( B_i^{\text{init}} \in [0, 1] \) for Player \( i \). A strategy \( f \) is legal for Player \( i \) with respect to \( B_i^{\text{init}} \) if for every \( v_0 \in V \) and strategy \( g \) for the other player, Player \( i \)’s bid in a finite outcome \( \pi = \text{out}(v_0, f, g) \) does not surpass his budget. Thus, for \( (b, v) = f(\pi) \), we have \( b \leq B_i^{\text{init}} - B_i(\pi) \).

**Richman games and threshold budgets**

The simplest qualitative objective is reachability: Player 1 has a target vertex \( v_R \) and an infinite outcome is winning for him if it visits \( v_R \). Reachability bidding games are known as Richman games [36, 35]. In Richman games both players have a target, which we denote by \( v_R \) and \( v_S \). The game ends once one of the targets is reached. Note that this definition is slightly different from standard reachability games since there, Player 2 has no target and his goal is to keep the game from \( v_R \). Though, we show that for our purposes, since Richman games have no ties, reachability games are equivalent to Richman games (see Lemma 4).

The central question that is studied on bidding games regards a threshold budget. A threshold budget is a function \( \text{Thresh} : V \rightarrow [0, 1] \) such that if Player 1’s budget exceeds \( \text{Thresh}(v) \) at a vertex \( v \), then he has a strategy to win the game. On the other hand, if Player 2’s budget exceeds \( 1 - \text{Thresh}(v) \), he can win the game. We sometimes use \( \text{Thresh}_1(v) \) to refer to \( \text{Thresh}(v) \) and \( \text{Thresh}_2(v) \) to refer to \( 1 - \text{Thresh}(v) \). We formalize the problem of finding threshold budgets as a decision problem. We define the \( \text{ThreshBUDG} \) problem, which takes as input a bidding game \( \mathcal{G} \), a vertex \( v \), and a value \( t \in [0, 1] \), and the goal is to decide whether \( \text{Thresh}(v) = t \).

Threshold values are shown to exist in [36] as well as how to compute them. We review briefly their results. Consider a Richman game \( \mathcal{G} = (V, E, v_R, v_S) \). We define the Richman function as follows. We first define \( R(v, i) \), for \( i \in \mathbb{N} \cup \{0\} \), where the intuition is that if Player 1’s budget exceeds \( R(v, i) \), he can win in at most \( i \) steps. We define \( R(v, 0) = 0 \) and \( R(v, 0) = 1 \) for every other vertex \( v \in V \). Indeed, Player 1 can win in 0 steps from \( v_R \) no matter what his initial budget is, and even if he has all the budget, he cannot win in 0 steps from anywhere else. Consider \( i \in \mathbb{N} \) and \( v \in V \). We denote by \( \text{adj}(v) \subseteq V \), the adjacent vertices to \( v \), so \( u \in \text{adj}(v) \) iff \( E(v, u) \). Let \( v^+ \) be the vertex that maximizes the expression \( \max_{a \in \text{adj}(v)} R(u, i - 1) \), and let \( v^- \) be the vertex that minimizes the expression \( \min_{a \in \text{adj}(v)} R(u, i - 1) \). We define \( R(v, i) = \frac{1}{2}(R(v^+, i - 1) + R(v^-, i - 1)) \). We define \( R(v) = \lim_{i \to \infty} R(v, i) \). The following theorem shows that \( R(v) \) equals \( \text{Thresh}(v) \), and throughout the paper we use them interchangeably. We give the proof of the theorem for completeness.

**Theorem 3.** [36] For every \( v \in V \), we have \( \text{Thresh}(v) = R(v) \), thus if Player 1’s budget at \( v \) exceeds \( R(v) \), he can win from \( v \), and if Player 2’s budget exceeds \( 1 - R(v) \), he can win from \( v \).

**Proof.** We prove for Player 1 and the proof for Player 2 is dual. Let \( t \in \mathbb{N} \) be an index such that \( B_1^{\text{init}} > R(v, t) \). We prove by induction on \( t \) that Player 1 wins in at most \( t \) steps. The base case is easy. For the inductive step, assume Player 1 has a budget of \( R(v, i) + \epsilon \). He bids \( b_1 = \frac{1}{2}(R(v^+, i - 1) - R(v^-, i - 1)) \). If he wins the bidding, he proceeds to \( v^- \) with a budget of \( R(v^-, i - 1) + \epsilon \). If he loses, then Player 2’s bid exceeds \( b_1 \) and the worst he can do is move to \( v^+ \). But then Player 1’s budget is at least \( R(v^+, i - 1) + \epsilon \). By the induction hypothesis, Player 1 wins in at most \( i - 1 \) steps from both positions. ▶
We make precise the equivalence between reachability and Richman games.

Lemma 4. Consider a bidding reachability game $G = (V, E, T)$, where $T \subseteq V$ is a target set of vertices for Player 1. Let $S \subseteq V$ be the vertices with no path to $T$. Consider the Richman game $G' = (V \cup \{v_R, v_S\}, E', v_R, v_S)$, where $E' = E \cup \{(v, v_R) : v \in T\} \cup \{(v, v_S) : v \in S\}$. For every $v \in V$, the threshold budget of $v$ in $G$ equals the threshold budget of $v$ in $G'$.

Finding threshold budgets

The authors in [35] study the complexity of threshold-budget problem and show that is in NP. They guess, for each vertex $v$ its neighbors $v^-$ and $v^+$, and devise a linear program with the constraints $R(v) = \frac{1}{2}(R(v^-) + R(v^+))$ and, for every neighbor $v'$ of $v$, we have $R(v^-) \leq R(v') \leq R(v^+)$. The program has a solution iff the guess is correct. They leave open the problem of determining the exact complexity of finding the threshold budgets, and they explicitly state that it is not known whether the problem is in P or NP-hard.

We improve on their result by showing that THRESH-BUDG is in NP and coNP. Our reduction uses an important observation that is made in [36], which will be useful later on. They connect between threshold budgets and reachability probabilities in Markov chains.

Observation 5. Consider a Richman game $G = (V, E, v_R, v_S)$. Let $M(G)$ be a Markov chain in which for each vertex $v \in V$, the probability of the edges $(v, v^+)$ and $(v, v^-)$ is $\frac{1}{2}$ and the other outgoing edges from $v$ have probability 0. Then, since $R(v) = \frac{1}{2}(R(v^+) + R(v^-))$, in $M(G)$, the probability of reaching $v_R$ from $v$ is $\text{THRESH}(v)$.

We reduce THRESH-BUDG to the problem of “solving” a simple stochastic game (SSG, for short) [22]. An SSG has two players; one tries to minimize the probability that the target is reached, and the second player tries to minimize it. It is well-known that the game has a value, which is the probability of reaching the target when both players play optimally. The problem of finding the value of an SSG is known to be in NP $\cap$ coNP. The SSG we construct can be seen as a turn-based game in which the player whose turn it is to move is chosen uniformly at random. The details of the proof can be found in the full version.

Theorem 6. THRESH-BUDG for Richman games is in NP $\cap$ coNP.

We stress the fact that the strategies in SSGs are very different from bidding games. As mentioned above, there, the strategies only prescribe which vertex to move the token to, whereas in bidding games, a strategy also prescribes what the next bid should be. So, a solution of a Richman game by reducing it to an SSG is existential in nature and does not give insight on the bids a player uses in his winning strategy. We will return to this point later on.

Objectives

We study zero-sum games. The qualitative games we focus on are parity games. A parity game is a triple $(V, E, p)$, where $p : V \to \{0, \ldots, d\}$ is a parity function that assigns to each vertex a parity index. An infinite outcome is winning for Player 1 iff the maximal index that is visited infinitely often is odd. The quantitative games we focus on are mean-payoff games.

A mean-payoff game is $(V, E, w)$, where $w : V \to \mathbb{Z}$ is a weight function on the vertices. We often refer to the sum of weights in a path as its energy. Consider an infinite outcome $\pi = v_0, (v_1, b_1, i_1), \ldots$. For $n \geq 0$, we use $\pi^n$ to refer to the prefix of length $n$ of $\pi$. The energy of $\pi^n$, denoted $E(\pi^n)$, is $\sum_{0 \leq i < n-1} w(v_i)$. We define the mean-payoff value of $\pi$ to be $\liminf_{n \to \infty} \frac{E(\pi^n)}{n}$. The value of $\pi$ can be thought of as the amount of money Player 1 pays.
Player 2. Note that the mean-payoff values do not affect the budgets of the players. That is, the game has two currencies: a “monopoly money” that is used to determine who moves the token and which the players do not care about once the game ends, and the mean-payoff value that is determined according to the weights of the vertices, which is the value that Min and Max seek to minimize and maximize, respectively. Consider a finite outcome \( \pi \). We use \( B_m(\pi) \) and \( B_M(\pi) \) to denote the sum of payments of Min and Max in the bids. Throughout the paper we use \( m \) and \( M \) to refer to Min and Max, respectively.

Strategy complexity
Recall that a winning strategy in a two-player game often corresponds to a system implementation. Thus, we often search for strategies that use limited or no memory. That is, we ask whether a player can win even with a memoryless strategy, which is a strategy in which the action depends only on the position of the game and not on the history. For example, in turn-based games, for \( i \in \{1, 2\} \), a memoryless strategy for Player \( i \) prescribes, for each vertex \( v \in V_i \), a successor vertex \( u \). It is well known that memoryless strategies are sufficient for winning in a wide variety of games, including turn-based parity games and turn-based mean-payoff games. In Richman games, the threshold budgets tell us who the winner of the game is. But, they do not give insight on how the game is won, namely what are the bids the winning player bids in order to win. Particularly, when the threshold budgets are 0 as we shall see in Lemmas 7 and 12.

We extend the definition of memoryless strategies to bidding games, though the right definition is not immediate. One can define a memoryless strategy as a function from vertex and budget to action (i.e., bid and vertex) similar to the definition in other games. However, this definition does not preserve the philosophy of implementation with no additional memory. Indeed, recall the proof of Theorem 3. One can define a strategy that, given a vertex \( v \in V \) and a budget \( B \), bids according to \( R_t(v) \), where \( t \) is the minimal index such that \( R_t(v) < B \). Clearly, the memory that is needed to implement such a strategy is infinite.

To overcome this issue, we use a different definition. We define a memoryless strategy in a vertex \( v \in V \) with initial budget \( B \in [0, 1] \) as a pair \( \langle u, f_B^v \rangle \), where \( u \in \text{adj}(v) \) is the vertex to proceed to upon winning and \( f_B^v : [0, 1] \to [0, 1] \) is a function that takes the current budget and, in mean-payoff games, also the energy, and returns a bid. We require that \( f_B^v \) is simple, namely a polynomial or a selection between a constant number of polynomials. For simplicity, we assume a memoryless strategy is generated for an initial vertex with an initial budget, thus there can be different strategies depending where the game starts and with what budget. Also, we call a concatenation of memoryless strategies, a memoryless strategy.

3 Parity Bidding Games
We study threshold budgets in bidding parity games. We first study strongly-connected parity games and show a classification for them; either Player 1 wins with every initial budget or Player 2 wins with every initial budget.

▶ Lemma 7. Consider a strongly-connected parity game \( G = (V, E, p) \). There exists \( \tau \in \{0, 1\} \) such that for every \( v \in V \), we have \( R(v) = \tau \). Moreover, we have \( \tau = 0 \) if \( \max_{v \in V} p(v) \) is odd.

Proof. The proof relies on the following claim: Player 1 wins a Richman game in which only his target is reachable, with every initial budget. The claim clearly implies the lemma as we
view a strongly-connected bidding parity game as a Richman game in which Player 1 tries to force the game to the vertex with the highest parity index, and Player 2 has no target, thus Player 1 wins with every initial budget. The claim is similar for Player 2. The proof of the claim follows from the fact that the threshold budget of a vertex $v \in V$ is some average between $\text{Thresh}(v_R)$ and $\text{Thresh}(v_S)$, and the average depends on the distances of $v$ to the two targets. When only Player 1’s target is reachable, we have $\text{Thresh}(v) = 0$. The details of the proof can be found in the full version.

Consider a bidding parity game $G = (V, E, p)$. Let $R$ and $S$ be the set of vertices in the BSCCs that are winning for Player 1 and Player 2, respectively. Let $G'$ be the Richman game that is obtained from $G$ by setting the target of Player 1 to be the vertices in $R$ and the target of Player 2 to be the vertices in $S$. The following lemma follows from Lemma 7.

Lemma 8. For every $v \in V$, we have that $\text{Thresh}(v)$ in $G$ equals $\text{Thresh}(v)$ in $G'$.

Lemma 8 allows us to obtain the positive results of Richman games in parity bidding games. In the full version, we construct memoryless strategies in Richman games. The idea is to show that a bid at a vertex $v$ of the form $\frac{R(v^+)-R(v^-)}{2} + \epsilon$ guarantees that either $v_R$ is reached within $|V|$ steps, or Player 1’s budget increases by a constant. Thus, we have the following.

Theorem 9. The threshold budgets in parity bidding games exist, are unique, $\text{THRESH-BUDG}$ is in NP \ intersection \ coNP, and memoryless strategies suffice for winning.

4 Mean-Payoff Bidding Games

We proceed to study mean-payoff games. We adjust the definition of threshold budgets to the quantitative setting.

Definition 10. Consider a mean-payoff bidding game $G = (V, E, w)$. The threshold budget in a vertex $v \in V$, denoted $\text{Thresh}(v)$, is a value $t \in [0, 1]$ such that

1. If Min’s budget exceeds $t$ at $v$, then he can guarantee a non-positive mean-payoff value,
   and
2. if Max’s budget exceeds $1 - t$, then he can guarantee a strictly positive value.

4.1 Solving Bidding Mean-Payoff Games

In this section we solve the problem of finding threshold values in bidding mean-payoff games. Our solution relies on work on probabilistic models, namely one-counter simple stochastic games [14, 13], and it is existential in nature. Namely, knowing what the threshold budget is in $v$ does not give much insight on how Min guarantees a non-negative value even if he has sufficient budget, and similarly for Max. Constructing concrete memoryless strategies for the two players is much more challenging and we show constructions in the following sections.

Recall that in bidding parity games, we showed a classification for strongly-connected games; namely, the threshold budgets in all vertices are in $\{0, 1\}$, thus either Player 1 wins with every initial budget or Player 2 wins with every initial budget. We show a similar classification for strongly-connected bidding mean-payoff games: the threshold budgets in all vertices of a strongly-connected bidding mean-payoff game are in $\{0, 1\}$, thus in a strongly-connected bidding mean-payoff game, for every initial energy and every initial budget, either
Min can guarantee a non-positive mean-payoff value or Max can guarantee a positive mean-payoff value. The classification uses a generalization of the Richman function to weighted graphs. Consider a strongly-connected bidding mean-payoff game $\mathcal{G} = (V, E, w)$ and a vertex $u \in V$. We construct a graph $\mathcal{G}^u = (V^u, E^u, w^u)$ by making two copies $u_s$ and $u_t$ of $u$, where $u_s$ has no incoming edges and $u_t$ has no outgoing edges. Thus, a path from $u_s$ to $u_t$ in $\mathcal{G}^u$ corresponds to a loop in $\mathcal{G}$. Recall that we denote by $w(v)$ the weight of the vertex $v$.

**Definition 11.** Consider a strongly-connected bidding mean-payoff game $\mathcal{G} = (V, E, w)$, a vertex $u \in V$. We define the *weighted Richman function* $W : V \rightarrow \mathbb{Q}$ first on $\mathcal{G}^u$. We define $W(u_t) = 0$ and for every $v \in (S \setminus \{u_t\})$, we define $W(v) = \frac{1}{2} (W(v^+) + W(v^-)) + w(v)$. In order to define $W$ on $\mathcal{G}$, we define $W(u)$ to be $W(u_s)$ in $\mathcal{G}^u$.

We use the connection with probabilistic models as in Observation 5 in order to show that $W$ is well defined. We view $\mathcal{G}^u$ as a rewarded Markov chain, in which, for $v \in V$, the outgoing edges from $v$ with positive probability probabilities are $\langle v, v^+ \rangle$ and $\langle v, v^- \rangle$, and their probability is $1/2$. The function $W$ coincides with the expected reward of a run that starts and returns to $u$, which in turn is well-defined since the probability of returning to $u$ is 1.

Similarly to the connection we show in Theorem 6 between Richman values and reachability probabilities in a simple-stochastic game, we prove Lemma 12 by connecting the threshold value in bidding mean-payoff games to the probability that a counter in a one-counter simple-stochastic games reaches value 0. We then use results from [14, 13] on this model to prove the lemma. The proof can be found in the full version.

**Lemma 12.** Consider a strongly-connected bidding mean-payoff game $\mathcal{G} = (V, E, w)$. There is $\tau \in \{0, 1\}$ such that for every $v \in V$, we have $\text{THRESH}(v) = \tau$. Moreover, we have $\tau = 0$ iff there exists $u \in V$ with $W(u) \leq 0$.

Lemma 13, which is also helpful in the following sections, shows how to connect the mean-payoff value with the objective of reaching energy 0 or maintaining non-negative energy. Its proof can be found in the full version.

**Lemma 13.** Consider a strongly-connected bidding mean-payoff game $\mathcal{G}$ and a vertex $u$ in $\mathcal{G}$.

- Suppose that for every initial budget and initial energy, Min has a strategy $f_m$, and there is a constant $N \in \mathbb{N}$ such that for every Max strategy $f_M$, a finite outcome $\pi = \text{out}(u, f_m, f_M)$ either reaches energy 0 or the energy is bounded by $N$ throughout $\pi$. Then, Min can guarantee a non-positive mean-payoff value in $\mathcal{G}$.

- If for every initial budget $B_M^\text{init} \in [0, 1]$ for Max there exists an initial energy level $n \in \mathbb{N}$ such that Max can guarantee a non-negative energy level in $\mathcal{G}$, then Max can guarantee a positive mean-payoff value in $\mathcal{G}$.

The proof of the following theorem can be found in the full version. Deciding the classification in Lemma 12 can be done in NP and coNP by guessing the neighbors the vertices and using linear programming, similarly to Richman games. Then, we reduce bidding mean-payoff games to Richman games in a similar way to the proof of Lemma 8 for parity games.

**Theorem 14.** Threshold budgets exist in bidding mean-payoff games, they are unique, and $\text{THRESH-BUDG}$ for bidding mean-payoff games is in $\text{NP} \cap \text{coNP.}$
4.2 A Memoryless Optimal Strategy for Min

We turn to the more challenging task of finding memoryless strategies for the players, and in this section we focus on constructing a strategy for Min. Theorem 9 and Lemma 12 allow us to focus on strongly-connected bidding mean-payoff games. Consider a strongly-connected bidding mean-payoff game $G = (V, E, w)$ that has a vertex $u \in V$ with $W(u) \leq 0$. We construct a Min memoryless strategy that guarantees that for every initial energy and every initial budget, either the energy level reaches 0 or it is bounded. By Lemma 13, this suffices for Min to guarantee a non-positive mean-payoff value in $G$.

The idea behind our construction is to tie between changes in the energy level and changes of the budget. That is, in order to decrease the energy by one unit, Min needs to invest at most one unit of budget (with an appropriate normalization), and when Max increases the energy by one unit, Min’s gain is at least one unit of budget. Our solution builds on an alternative solution to the two-loop game in Figure 1. This solution is inspired by a similar solution in [35].

Example 15. Consider the bidding mean-payoff game that is depicted in Figure 1. We show a Min strategy that guarantees a non-positive mean-payoff value. Consider an initial budget of $B^\text{init}_m \in [0, 1]$ and an initial energy level of $k_1 \in \mathbb{N}$. Let $N \in \mathbb{N}$ be such that $B^\text{init}_m > k N$. Min bids $\frac{k}{N}$ and takes the $(-1)$-weighted edge upon winning. Intuitively, Min invests $\frac{k}{N}$ for every decrease of unit of energy and, since by losing a bidding he gains at least $\frac{k}{N}$, this is also the amount he gains when the energy increases. Formally, it is not hard to show that the following invariant is maintained: if the energy level reaches $k \in \mathbb{N}$, Min’s budget is at least $\frac{k}{N}$. Note that the invariant implies that either an energy level of 0 is reached infinitely often, or the energy is bounded by $N$. Indeed, in order to cross an energy of $N$, Max would need to invest a budget of more than 1. Lemma 13 implies that the mean-payoff value is non-positive, and we are done.

Extending this result to general strongly connected games is not immediate. Consider a strongly-connected game $G = (V, E, w)$ and a vertex $u \in V$. We would like to maintain the invariant that upon reaching $u$ with energy $k$, the budget of Min exceeds $k/N$, for a carefully chosen $N$. The game in the simple example above has two favorable properties that general SCCs do not necessarily have. First, unlike the game in the example, there can be infinite paths that avoid $u$, thus Min might need to invest budget in drawing the game back to $u$. Moreover, different paths from $u$ to itself may have different energy levels, so bidding a uniform value (like the $\frac{k}{N}$ above) is not possible. The solution to these problems is surprisingly elegant and uses the weighted Richman function in Definition 11.

Consider an initial budget of $B^\text{init}_m \in [0, 1]$ for Min and an initial energy $k_1 \in \mathbb{N}$. We describe Min’s strategy $f_m$. At a vertex $v \in V$, Min’s bid is $w(v) = \frac{W(v) - W(v^-)}{2} \cdot \frac{1}{N}$ and he proceeds to $v^-$ upon winning, where we choose $N \in \mathbb{N}$ in the following. Let $w_M$ be the maximal absolute weighted Richman value in $G$, thus $w_M = \max_{v \in V} |W(v)|$. Let $b_M$ be the maximal absolute “bid”, thus $b_M = \max_{v \in V} \left| \frac{W(v) - W(v^-)}{2} \right|$. We choose $N \in \mathbb{N}$ such that $B^\text{init}_m > \frac{k_1 + b_M + w_M}{N}$.

In the following lemmas we prove that $f_m$ guarantees that an outcome either reaches energy level 0 or that the energy is bounded, as well as showing that $f_m$ is legal, i.e., that Min always bids less than his budget. The following lemma is the crux of the construction as it connects the weighted Richman function with the change in energy and in budget. Recall that for a finite outcome $\pi$ the accumulated energy in $\pi$ is $E(\pi)$ and the payments of Min throughout $\pi$ is $B_m(\pi)$.\[CONCUR 2017\]
21:12 Infinite-Duration Bidding Games

Lemma 16. Consider a Max strategy $f_M$, and let $\pi = \text{out}(f_m, f_M)$ be a finite outcome that starts in a vertex $v$ and ends in $v'$. Then, we have $W(v) - W(v') \geq E(\pi) + N \cdot B_m(\pi)$.

Proof. We prove by induction on the length of $\pi$. In the base case $v = v'$, thus $E(\pi) = B_m(\pi) = 0$ and the claim is trivial. For the induction step, let $b$ be the winning bid in the first round and let $\pi'$ be the suffix of $\pi$ after the first bidding. We distinguish between two cases. In the first case, Min wins the bidding, pays $b = \frac{W(v') - W(v')}{2}$, and we are done. Thus, we have $E(\pi) + N \cdot B_m(\pi) = w(v) + E(\pi') + N(b + B_m(\pi'))$. By the induction hypothesis, we have $E(\pi') + N \cdot B_m(\pi') \leq W(v') - W(v')$, thus $E(\pi) + N \cdot B_m(\pi) \leq w(v) + \frac{W(v') - W(v')}{2} + W(v') - W(v') = W(v) - W(v')$, and we are done. For the second case, suppose Max wins the bidding. Min’s gain is $-b < -\frac{W(v') - W(v')}{2}$, and Max proceeds to $v''$ having $W(v'') \leq W(v')$. Similar to the previous case, we have $E(\pi) + N \cdot B_m(\pi) = w(v) + E(\pi') + N(-b + B_m(\pi')) \leq w(v) - \frac{W(v') - W(v')}{2} + W(v') - W(v') = W(v) - W(v')$, and we are done. ▶

The following corollary of Lemma 16 explains why we refer to our technique as “tying energy and budget”. Its proof follows from the fact that $W(u_i) \leq 0$ and $W(u_i) = 0$.

Corollary 17. Consider a Max strategy $f_M$, and let $\pi = \text{out}(f_m, f_M)$ be a finite outcome from $u$ to $u$. Then, we have $-N \cdot B_m(\pi) \geq E(\pi)$.

We formalize the intuition above by means of an invariant that is maintained throughout the outcome. Recall that the game starts from a vertex $u \in V$ with $W(u) \leq 0$, the initial energy is $k_1 \in \mathbb{N}$, Min’s initial budget is $B_m^{\text{init}} \in [0, 1]$, and $N$ is such that $B_m^{\text{init}} > \frac{k_1 + b_M + w_M}{N}$.

Lemma 18. Consider a Max strategy $f_M$, and let $\pi = \text{out}(f_m, f_M)$ be a finite outcome. Then, when the energy level reaches $k$, Min’s budget is at least $\frac{k_1 + b_M + w_M}{N}$.

Proof. The invariant clearly holds initially. Consider a partition $\pi = \pi_1 \cdot \pi_2$, where $\pi_1$ is a maximal prefix of $\pi$ that ends in $u$ and $\pi_2$ starts in $u$ and ends in a vertex $v \in V$. The energy level at the end of $\pi$ is $k = k_1 + E(\pi)$. Recall that $B_m(\pi)$ is the sum of Min’s payments in $\pi$, thus his budget at the end of $\pi$ is $B_m^{\text{init}} - (B_m(\pi_1) + B_m(\pi_2))$. By Corollary 17, we have $-B_m(\pi_1) \geq \frac{1}{N}E(\pi_1)$ and by Lemma 16, we have $-B_m(\pi_2) \geq \frac{1}{N}(E(\pi_2) - W(u) + W(v)) \geq \frac{1}{N}(E(\pi_2) - 0 - w_M)$. Combining with $B_m^{\text{init}} \geq \frac{k_1 + b_M + w_M}{N}$, we have that the new budget is at least $\frac{k_1 + b_M + w_M}{N} + \frac{E(\pi_1)}{N} + \frac{E(\pi_2) - w_M}{N} = \frac{k_1 + b_M}{N}$, and we are done. ▶

Lemma 18 implies that Min always has sufficient budget to bid according to $f_m$, thus the strategy is legal. Moreover, since Min’s budget cannot exceed 1, Lemma 18 implies that if the energy does not reach 0, then it is bounded by $N - b_M$. Thus, Lemma 13 implies that Min has a memoryless strategy that guarantees a non-positive mean-payoff value in a strongly-connected bidding mean-payoff game having a vertex $u$ with $W(u) \leq 0$. Combining with the memoryless strategy in parity games, we have the following.

Theorem 19. Consider a bidding mean-payoff game $G = (V, E, w)$ and a vertex $v \in V$. If Min’s initial budget exceeds $\text{Thresh}(v)$, he has a memoryless strategy that guarantees a non-positive mean-payoff value.

4.3 A Memoryless Optimal Strategy for Max

The complementary result of the previous section is more involved. Consider a strongly-connected bidding mean-payoff game $G$ with a vertex $u$ that has $W(u) > 0$. We devise a Max strategy that guarantees a positive mean-payoff value in $G$. We start with a fragment...
of the general case called recurrent SCCs, and we generalize our solution later. We say that an SCC $G = (V, E)$ is a recurrent, if there is a vertex $u \in V$ such that every cycle in $G$ includes $u$. We refer to $u$ as the root of $G$.

Intuitively, the construction has two ingredients. First, we develop the idea of tying energy and budget. We construct a Max strategy $f_M$ that guarantees the following: when Max invests a unit of budget (with an appropriate normalization), then the energy increases by at least one unit, and when the energy decreases by one unit, Max’s gain is at least $z > 1$ units of budget, where $z$ arises from the game graph. The second ingredient concerns the normalization factor. Recall that in the previous section it was a constant $\frac{1}{M}$. Here on the other hand, it cannot be constant. Indeed, if the normalization does not decrease as the energy increases, Max’s budget will eventually run out, which is problematic since with a budget of 1, Min can guarantee reaching energy level 0, no matter how high the energy is.

The challenge is to decide when and how to decrease the normalization factor. We split $\mathbb{N}$ into energy blocks of size $M$, for a carefully chosen $M \in \mathbb{N}$. The normalization factor of the bids depends on the block in which the energy is in, and we refer to it as the currency of the block. The currency of the $n$-th block is $z^{-n}$. Note that the currency of the $(n - 1)$-th block is higher by a factor of $z$ from the currency of the $n$-th block. This is where the first ingredient comes in: investing in the $n$-th block is done in the currency of the $n$-th block, whereas gaining in the $n$-th block is in the higher currency of the $(n - 1)$-th block. We switch between the currencies when the energy moves between energy blocks only at the root $u$ of $G$. This is possible since $G$ is a recurrent SCC. The mismatch between gaining and investing is handy when switching between currencies as we cannot guarantee that when we reach $u$ the energy is exactly in the boundary of an energy block.

We formalize this intuition. We start by finding an alternative definition for the weighted Richman function. Recall that in order to define $W$, we constructed a new graph $G^w$ by splitting $u$ into $u_s$ and $u_t$. We define the contribution of a vertex $v \in V$ to $W(u_s)$, denoted $cont(v)$, as follows. We have $cont(u_s) = 1$. For a vertex $v \in V$, we define $pre(v) = \{v' \in V : v = v'^{-}\text{ or } v = v'^{+}\}$. For $v \in V$, we define $cont(v) = \sum_{v' \in pre(v)} \frac{1}{2} \cdot cont(v')$. The proof of the following lemma uses the connection with probabilistic models, and follows from standard arguments there.

Lemma 20. We have $W(u) = \sum_{v \in V} (cont(v) \cdot w(v))$.

Let $z = (\sum_{v \in V} \sum_{w(v) \geq 0} cont(v) \cdot w(v)) / (\sum_{v \in V} w(v) - \sum_{v \in V} \sum_{w(v) < 0} cont(v) \cdot |w(v)|)^{-1}$. Since $W(u) > 0$, we have $z > 1$. Let $G^w$ be the game that is obtained from $G$ by multiplying the negative-weighted vertices by $z$, thus $G^w = (V, E, w^z)$, where $w^z(v) = w(v)$ if $w(v) \geq 0$ and otherwise $w^z(v) = z \cdot w(v)$. We denote by $W^z$ the weighted threshold budgets in $G^w$. The following lemma follows immediately from Lemma 20.

Lemma 21. We have $W^z(u) = 0$.

We define the partition into energy blocks. Let cycles$(u)$ be the set of simple cycles from $u$ to itself and $w_M = \max_{u \in cycles(u)} |E(u)|$. We choose $M$ such that $M \geq (b_M + 3w_M)/(1 - z^{-1})$, where $b_M$ is the maximal bid as in the previous section. We partition $\mathbb{N}$ into blocks of size $M$. For $n \geq 1$, we refer to the $n$-th block as $M_n$, and we have $M_n = \{M(n - 1), M(n - 1) + 1, \ldots, Mn - 1\}$. We use $\beta^u_n$ and $\beta^u_n$ to mark the upper and lower boundaries of $M_n$, respectively. We use a $M_{\geq n}$ to denote the set $\{M_n, M_{n+1}, \ldots\}$. Consider a finite outcome $\pi$ that ends in $u$ and let visits$_\pi(u)$ be the set of indices in which $\pi$ visits $u$. Let $k_j \in \mathbb{N}$ be an initial energy. We say that $\pi$ visits $M_n$ if $k_j + E(\pi) \in M_n$. We say that $\pi$ stays in $M_n$ starting from an index $1 \leq i \leq |\pi|$ if for all $j \in visits_\pi(u)$ such that $j \geq i$, we have $k_j + E(\pi_1, \ldots, \pi_j) \in M_n$. 

CONCUR 2017
We are ready to describe Max’s strategy $f_M$. Suppose the game reaches a vertex $v$ and the
energy in the last visit to $v$ was in $M_n$, for $n \geq 1$. Then, Max bids $z^{-n-1} \cdot \mathbb{1}(W^z(v^+) - W^z(v^-))$
and proceeds to $v^+$ upon winning. Note that currency changes occur only in $u$. Recall that
for an outcome $\pi$, the sum of payments of Max is $B_M(\pi)$ and let $E^z(\pi)$ be the change in
energy in $G^z$. The proof of Lemma 16 can easily be adjusted to this setting.

\textbf{Lemma 22.} Consider a Min strategy $f_m$, and let $\pi = \text{out}(f_m, f_M)$ be a finite outcome that
starts in $v$, ends in $v'$, and stays within a block $M_n$, for $n \geq 1$. We have $W^z(v) - W^z(v') \leq E^z(\pi) - z^n \cdot B_M(\pi)$. In particular, for $\pi \in \text{cycles}(u)$, we have $E^z(\pi) \leq z^n \cdot B_M(\pi)$.

We relate between the changes in energy in the two structures. The proof of the following
lemma can be found in the full version.

\textbf{Lemma 23.} Consider an outcome $\pi \in \text{cycles}(u)$. Then, $E(\pi) \geq E^z(\pi)$ and $E(\pi) \geq zE^z(\pi)$.

A corollary of Lemmas 22 and 23 is the following. Recall that $B_M(\pi)$ is the amount that
Max pays, thus it is negative when Max gains budget. Intuitively, the corollary states that
for an outcome $\pi$ such that $\sum_{i} z_i E^z(\pi)$ decreases, he gains in the currency of
energy in the last visit to $\pi$. Consider an outcome $\pi \in \text{cycles}(u)$. We have $E^z(\pi) \leq z^n \cdot B_M(\pi)$.

\textbf{Corollary 24.} Consider a Min strategy $f_m$, and let $\pi = \text{out}(f_m, f_M)$ be a finite outcome
such that $\pi \in \text{cycles}(u)$. Then, we have $E(\pi) \geq z^n \cdot B_M(\pi)$ and $zE^z(\pi) \geq z^n \cdot B_M(\pi)$.

Consider an initial Max budget $B_M^{\text{init}} \in [0, 1]$. We choose an initial energy $k_l \in \mathbb{N}$ with
which $f_M$ guarantees that energy level 0 is never reached. Recall the intuition that increasing
the energy by a unit requires an investment of a unit of budget in the right currency. Thus,
increasing the energy from the lower boundary $\beta_n^1$ of $M_n$ to its upper boundary $\beta_n^1$, costs
$\beta_n^1 \cdot z^{-n}$. We use $\text{cost}(M_n)$ to refer to $M \cdot z^{-n}$ and $\text{cost}(M_{\geq n}) = \sum_{i=n}^{\infty} \text{cost}(M_i)$. A first
guess for $k_l$ would be $\beta_n^1$ such that $B_M^{\text{init}} > \text{cost}(M_{\geq n})$. This is almost correct. We need some wiggle room to allow for changes in the currency. Let $w_M = 2w_M + b_M$, where recall that $w_M = \max_{\pi \in \text{cycles}(u)} E(\pi)$ and that $b_M$ is the maximal bid. We define $k_l$ to be
$\beta_n^1$ such that $B_M^{\text{init}} > w_M \cdot z^{-(n-1)} + \text{cost}(M_{\geq n})$ and $\sum_{i=n}^{\infty} \text{cost}(M_i) > 1$, thus Min cannot
decrease the energy to 0.

Consider a Min strategy $f_m$, and let $\pi = \text{out}(f_m, f_M)$ be a finite outcome. We partition
$\pi$ into subsequences in which the same currency is used. Let $\pi = \pi_1 \cdot \pi_2 \cdots \cdot \pi_l$ be a partition of $\pi$. For $1 \leq i \leq l$, we use $\pi^i$ to refer to the prefix $\pi_1 \cdot \cdots \cdot \pi_i$ of $\pi$, and we use $e^i = k_l + E(\pi^i)$
to refer to the energy at the end of $\pi^i$. Consider the partition in which, for $1 \leq i \leq l$, the
prefix $\pi^i$ visits $u$ and $\pi_i$ is a maximal subsequence that stays in some energy block.

Suppose $\pi_i$ stays in $M_n$. There can be two options; either the energy decreases in $\pi_i$,
then the energy before it $e^{i-1}$ is in $M_{n+1}$ and the energy after it $e^i$ is in $M_n$, or it increases,
thus $e^{i-1} \in M_{n-1}$ and $e^i \in M_n$. We then call $\pi^i$ decreasing and increasing, respectively.
The definition of $w_M$ and the fact that $G$ is recurrent imply that upon entering $M_n$, the
energy is within $w_M$ of the boundary. Thus, in the case that $\pi^i$ is decreasing, the energy at
the end of $\pi^i$ is $e^i \geq \beta_n^1 - w_M$ and in the case it is increasing, we have $e^i \leq \beta_n^1 + w_M$. Let
$\ell_0 = 0$, and for $i \geq 1$, let $\ell_i = (\beta_n^{i+1} - w_M) - e^i$ in the first case and $\ell_i = (\beta_n^{i+1} + w_M) - e^i$
in the second case. Note that $\ell_i \in \{0, \ldots, 2w_M\}$. In the full version, we prove the following
invariant on Max’s budget when changing between energy blocks.

\textbf{Lemma 25.} For every $i \geq 0$, suppose $\pi^i$ ends in $M_n$. Then, Max’s budget is at least
$(w_M + \ell_i) \cdot z^{-n-1} + \text{cost}(M_{\geq n})$, where $\hat{n} = n + 1$ if $\pi^i$ is decreasing and $\hat{n} = n$ if $\pi^i$ is
increasing.
It is not hard to show that Lemma 25 implies that \( f_M \) is legal. That is, consider a finite outcome \( \pi \) that starts immediately after a change in currency. Using Lemma 22, we can prove by induction on the length of \( \pi \) that Max has sufficient budget for bidding. The harder case is when \( \pi \) decreases, and the proof follows from the fact that \( \text{wiggle} \) is in the higher currency of the lower block. Combining Lemma 25 with our choice of the initial energy, we get that the energy never reaches 0 as otherwise Min invests a budget of more than 1. Lemma 13 implies that Max guarantees a positive mean-payoff value in a strongly-connected game.

**Theorem 26.** Consider a bidding mean-payoff game \( G = \langle V, E, w \rangle \) in which all BSCCs are recurrent. For a vertex \( v \in V \), if Max has an initial budget that is greater than \( 1 - \text{Thresh}(v) \), he has a memoryless strategy that guarantees a positive mean-payoff value.

### General strongly-connected games

In the full version, we develop a constant-memory strategy for Max that guarantees a positive mean-payoff value. The difficulty lies in coping with outcomes in which the energy forms a sine-like wave on the boundary of an energy block. In recurrent SCCs, we can change currency every time the wave changes block, which does not work in general SCCs as we show in an example. We develop further the two ingredients that are used in \( f_M \). First, recall that investing in an energy block \( M_n \) is in the currency of the \( n \)-th block, whereas gaining is in the higher currency of the \( (n - 1) \)-th block. In general games, we need a stronger property: investing in \( M_n \) is in the lower currency of the \((n + 1)\)-th block while gaining is still in the higher currency of the \((n - 1)\)-th block. Next, we differentiate between even blocks, i.e., \( M_{2n} \), and odd blocks, i.e., \( M_{2n+1} \), for some \( n \in \mathbb{N} \). When the energy level reaches an even block \( M_{2n} \), the currency used is \( z^{-n} \). In order to determine the currency in the odd blocks, we take the history of the play into account; the currency matches the currency in the last energy block that was visited before entering \( M_{2n+1} \). Hence, we call our strategy a constant-memory strategy. The odd blocks serve as “buffers” so that when we change currency, there is a sufficiently large change in energy that in turn implies that Max’s budget sufficiently increases compared with the change in energy. Combining with the memoryless strategy in parity games of Theorem 9, we have the following.

**Theorem 27.** Consider a bidding mean-payoff game \( G = \langle V, E, w \rangle \). For a vertex \( v \in V \), if Max has an initial budget that is greater than \( 1 - \text{Thresh}(v) \), he has a constant-memoryless strategy that guarantees a positive mean-payoff value.

## Discussion and Future Directions

We introduce and study infinite-duration bidding games in which the players bid for the right to move the token. This work belongs to a line of works that transfer concepts and ideas between the areas of formal methods and algorithmic game theory (AGT, for short). Richman games originated in the game theory community in the 90s and recently gained interest by the AGT community [30]. We combine them with the study of infinite-duration games, which is well-studied in the formal methods community. Prior to this work, a series of works focused on applying concepts and ideas from formal methods to resource-allocation games [10, 8, 9, 5, 6, 34], which constitutes a well-studied class of games in AGT. More to the formal methods side, there are many works on games that share similar concepts to these that are studied in AGT. For example, logics for reasoning about multi-agent systems [3, 19, 39],
studies of equilibria in games related to synthesis and repair problems [18, 25, 1, 15], and studies of infinite-duration non-zero-sum games [21, 16, 17, 12].

There are several problems we left open as well as plenty of future research directions. We list a handful of them below. We showed that the complexity of THRESH-BUDG is in NP and coNP. We leave open the problem of determining its exact complexity. We conjecture that it is reducible from solving simple stochastic games, which will show that it is as hard as several other problems whose exact complexity is unknown. In this work we focused on parity and mean-payoff games. Energy games are games that are played on a weighted graph, where one of the players tries to reach negative energy and the second player tries to prevent it. Note that unlike parity and mean-payoff, the energy objective is not prefix independent. We can show that threshold budgets exist in energy games. The complexity of THRESH-BUDG in energy games is interesting and is tied with recent work on optimizing the probability of reaching a destination in a weighted MDP [26, 41]. For acyclic energy bidding games, the problem is PP-hard using a result in [26], and for a single-vertex games the problem is in P using the direct formula of [32]. For general games the problem is open.

Acknowledgments. We thank Petr Novotný for helpful discussions and pointers.

References

<table>
<thead>
<tr>
<th>Page</th>
<th>Reference</th>
</tr>
</thead>
</table>
On the Power of Name-Passing Communication*

Yuxi Fu

BASICS, Shanghai Jiao Tong University, China
fu-yx@cs.sjtu.edu.cn

Abstract
It is shown that generally higher order process calculi cannot be interpreted in name-passing calculi in a robust way.

1998 ACM Subject Classification F.1.1 Models of Computation, F.1.2 Modes of Computation

Keywords and phrases Interaction theory, expressiveness, bisimulation

1 Introduction
Many process calculi have been proposed [19]. It is important to investigate the relative expressiveness between them [29, 25, 18, 20, 23, 10, 8]. It is even more important to study the relative expressiveness between classes of these models [5]. Positive results can tell us which model plays a foundational role in interaction theory and which acts as a prototype in applications. A negative result often reveals a deep distinct property of a model we probably were not aware of beforehand.

There are basically three classes of interactively Turing powerful models characterized by the content of communication. In the name-passing calculi the messages are channels (channel names) [16, 28, 22]. The power of this class of process calculi crucially depends on the ability to pass around private channels. The general view is that the π-calculus is the “λ-calculus” of concurrency theory [15] and is capable of modelling all kinds of phenomena in both theory and application [34]. The second class contains higher order process calculi. The pure process-passing calculus is too weak [5]. The messages in higher order process calculi are normally abstractions [25, 26, 31, 32]. A receiving process can instantiate the parameters of the received abstraction with its own private channels. A seemingly exception is Thomsen’s CHOCS [29, 30]. However there is a very strong operator in CHOCS, the so-called relabelling operator [14]. If we think of it the relabelling operator can achieve the effect of instantiations of abstractions. So in CHOCS process-passing is abstraction-passing in disguise. Concerning the relative expressive power of the higher order process calculi, it is a popular belief that they can be completely interpreted in the name-passing calculi. If the messages are neither channels nor abstractions, they can always be coded up by natural numbers. The class of value-passing calculi consists of those models whose contents of communications are numbers [13, 14, 11, 12]. It is proved in [5] that value-passing calculi and name-passing calculi are incompatible in expressive power. This result appears somewhat surprising at first sight. The value-passing calculi turns out to be very expressive. A great deal of higher order communications can be interpreted by value-passing communications [3]. More precisely, if in a higher order process calculus messages may contain neither unbound private channels nor global channels, then the calculus is a submodel of the standard value-passing calculus [3]. The intuition is

* The support from NSFC (61472239, ANR 61261130589) is gratefully acknowledged.

© Yuxi Fu; licensed under Creative Commons License CC-BY
28th International Conference on Concurrency Theory (CONCUR 2017).
Editors: Roland Meyer and Uwe Nestmann; Article No. 22; pp. 22:1–22:15
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany
that such messages can be coded up by numbers. Therefore abstraction-passing becomes value-passing. Moreover the Gödel encoding is bijective. So full abstraction poses no problem. What we will show in this short paper, by generalizing the proof in [5], is that even such restricted higher order communications cannot be interpreted by the name-passing calculi in a faithful manner. This result contributes to a better understanding of the relative expressive power of the three classes of interaction models.

Section 2 introduces the criteria for relative expressiveness. Section 3 fixes notations for the \( \pi \)-calculus. Section 4 proves a general negative result about non-interpretablity in the \( \pi \)-calculus. Section 5 derives the main result. Section 6 makes some concluding comments.

## 2 Relative Expressiveness

Given two models \( M, N \), in what sense is \( N \) at least as expressive as \( M \)? Different researchers have different criteria. In logic terms the answer is given by submodel relationship. In what follows we shall motivate a set of criteria that will uncover our definition of submodel relationship.

Generally we understand an interpretation of \( M \) in \( N \) as a binary relation \( \propto \) from the set of \( M \)-processes to the set of \( N \)-processes. The interpretation should be total in the sense that for every \( M \)-process \( M \) there is an \( N \)-process \( N \) such that \( M \propto N \). The interpretation should not introduce extra divergence. There should not be an infinite internal action sequence \( N \xrightarrow{\tau} N_0 \xrightarrow{\tau} N_1 \xrightarrow{\tau} N_2 \xrightarrow{\tau} \ldots \) such that \( M \propto N \), for all \( i \in \omega \) if \( M \) cannot do any infinite internal action sequence. More formally this energy respect criterion is introduced in [24].

> **Definition 1.** The relation \( \propto \) is codivergent if \( M_0 \propto N_0 \) implies the following: (i) If there is an infinite internal action sequence \( N_0 \xrightarrow{\tau} N_1 \xrightarrow{\tau} N_2 \xrightarrow{\tau} \ldots \), there is some \( M' \) such that \( M \xrightarrow{\tau} M' \propto N_i \) for some \( i > 0 \). (ii) If \( M_0 \xrightarrow{\tau} M_1 \xrightarrow{\tau} M_2 \xrightarrow{\tau} \ldots \) is an infinite internal action sequence, there is some \( N' \) such that \( N \xrightarrow{\tau} N' \) and \( M_j \propto N' \) for some \( j > 0 \).

Codivergence is a computational property. The unique equality for computation, the extensional equality, is divergence sensitive. Codivergence is the divergence sensitive property reformulated in the presence of bisimulation. Bisimulation property [15, 21] is in essence also about computations [4]. This is particularly evident in the following definition, where \( \equiv \) is the reflexive and transitive closure of the one step internal transition \( \xrightarrow{\tau} \).

> **Definition 2.** The relation \( \propto \) is a bisimulation if the following are valid whenever \( M \propto N \):

1. If \( M \xrightarrow{\tau} M' \), then
   a. either \( N \equiv N' \) for some \( N' \) such that \( M \propto N' \) and \( M' \propto N' \), or
   b. \( N \equiv N'' \xrightarrow{\tau} N' \) for some \( N'', N' \) such that \( M \propto N'' \) and \( M' \propto N' \).
2. If \( N \xrightarrow{\tau} N' \), then
   a. either \( M \equiv M' \) for some \( M' \) such that \( M' \propto N \) and \( M' \propto N' \), or
   b. \( M \equiv M'' \xrightarrow{\tau} M' \) for some \( M'', M' \) such that \( M'' \propto N \) and \( M' \propto N' \).

Case (1a) and case (2a) are about deterministic computation whereas case (1b) and case (2b) are to do with nondeterministic computation. The reader must have noticed that this is the branching bisimulation defined in [33]. The reason for using branching bisimulation is that even if we allow the interpretation \( N \) of \( M \) to carry out some internal adjustments (actions), we do not allow \( N \) to introduce any extra nondeterministic computation step. If \( N \equiv N' \) for some \( N' \) not equivalent to \( N \), then \( N' \) cannot be an interpretation of \( M \). This would lead to a contradiction if \( M \) cannot do any nondeterministic computation. The problem with the weak bisimulation [14] is that it confuses nondeterministic computation with deterministic computation. This is a confusion that never happens in computation theory.
We also need criteria that take into account of interaction. We assume that all external actions admitted in all our models are carried out at global channels. We say that a process $P$ is observable at a global channel $a$ if $P \xrightarrow{} P'$ for some $P'$ such that $P'$ may perform an external action at channel $a$. If $N$ is an interpretation of $M$ then they should have the same capacity to communicate at any particular global channel, hence the following [17].

**Definition 3.** The relation $\propto$ is equipollent if $M \propto N$ implies that, for every global channel $a$, $M$ is observable at $a$ if and only if $N$ is observable at $a$.

If $M \propto N$ then there should be no difference between $M$ and $N$ detectable by equivalent observers from the respective models. In other words $\propto$ should be closed under the concurrent composition operator. Without this closure property it does not make sense to talk about relative expressiveness for interaction model.

**Definition 4.** The relation $\propto$ is extensional if $M \mid M' \propto N \mid N'$ whenever $M \propto N$ and $M' \propto N'$.

Intuitively $M$ is a submodel of $N$ if for each $M$-process $M$ there is an $N$-process $N$ that is equal to $M$ as it were. Obviously the equality on $M$-processes must be a special submodel relationship.

**Definition 5.** The absolute equality $=_{M}$ on $M$ is the largest binary relation on $M$-processes that renders true the following statements.

1. It is reflexive.
2. It is extensional, equipollent, codivergent, bisimilar.

We often omit the subscript in $=_{M}$. It is important that expressiveness and equality are defined in completely the same fashion. Otherwise it would not be a submodel relationship, confer Lemma 7.

**Definition 6.** A subbisimilarity from $M$ to $N$ is a binary relation $\propto$ from $M$-processes to $N$-processes such that the following statements are valid.

1. It is total and semantical, the latter means that $M' = M \propto N$ implies $M' \propto N$.
2. It is extensional, equipollent, codivergent, bisimilar.

The first condition of Definition 6 is a generalization of the reflexivity condition of Definition 5. Like reflexivity it is a preliminary requirement. The semantical condition asks for nothing more than that the relation $\propto$ should be syntax independent when seen as an interpretation from $M$ to $N$. We say that $M$ is a submodel of $N$, notation $M \subseteq N$, if there is a subbisimilarity from $M$ to $N$. The terminology is enforced by the following full abstraction lemma.

**Lemma 7.** Suppose $\propto$ is a subbisimilarity from $M$ to $N$, and $M \propto N$ and $M' \propto N'$. Then $M = M'$ if and only if $N = N'$.

**Proof.** $N =_{N} N'$ implies $M =_{M} M'$ since the composition $\propto; =_{N}; \propto^{-1}$ is a subbisimilarity, where $\propto^{-1}$ is the reverse relation of $\propto$. Conversely if $M =_{M} M'$ then $M' \propto N$ by the semantical condition. It follows from $M' \propto N'$ and $M' \propto N$ that $N =_{N} N'$ because $\propto; =_{N}$ is a subbisimilarity.

In the rest of the paper we say that $P \xrightarrow{\tau} P'$ is a deterministic computation step, notation $P \xrightarrow{\cdot} P'$, if $P' = P$, and that it is a nondeterministic computation step, notation $P \xrightarrow{\cdot} P'$, if $P' \neq P$. We will write $\xrightarrow{\cdot} (\rightarrow^*)$ for the (reflexive and) transitive closure of $\rightarrow$. A $\tau$-descendant of $T$ is a term $T'$ such that $T \xrightarrow{\cdot} T'$. Every model has an unobservable process $0$ whose every computation is finite and an unobservable process $\Omega$ whose every
computation is infinite. It is easy to see that both 0 and Ω can only do deterministic computation. For every subbisimilarity α it holds that Q = Ω whenever Ω ≀ Q or Q ≀ Ω. Similarly Q = 0 whenever 0 ≀ Q or Q ≀ 0.

For more motivations to the absolute equality and the subbisimilarity, consult [5].

3 Name-Passing Calculus

We fix in this section the target model [16]. Unlike in the original presentation we introduce syntactical distinction between global channels, private channels and channel variables.

= Let C_g be the set of global channels. The elements of C_g are denoted by a, b, c, d, e, f.
= Let C_p be the set of private channels. The elements of C_p are denoted by l, m, n, o, p, q.
= Let C_v be the set of channel variables. The elements of C_v are denoted by u, v, w, x, y, z.

The set C_g ∪ C_p ∪ C_v will be ranged over by μ, ν, and the set C_g ∪ C_p by α, β. The π-terms are constructed from the following grammar.

\[ S, T := \sum_{i \in I} \mu(x).T_i | \sum_{i \in I} \mu[\nu_i.T_i | S | T | (p)T | [\mu=\nu]T | [\mu\neq\nu]T | !\mu(x).T. \]

In the above definition I is a finite indexing set. We abbreviate \(\sum_{i \in I} \mu(x).T_i\) and \(\sum_{i \in I} \mu[\nu_i.T_i\) to 0 if I = ∅. If the size of I is one, we get input term \(\mu(x).T\) and output term \(\mu[\nu].T\). We abbreviate \((p)\mu.p.T\) to \(\mu(p).T\), \(a(x).T\) to \(a.T\) if x does not appear in T, and \(\pi(q).T\) to \(\pi.T\) if q does not appear in T. The labeled transition semantics is defined by the following rules.

\[ \sum_{i \in I} \alpha(x).T_i \xrightarrow{\alpha} T_i(\beta/x) \]

\[ T \xrightarrow{\alpha} T' \quad p \notin \lambda \]

\[ S | T \xrightarrow{\alpha} S' | T' \]

\[ S | T \xrightarrow{(p)T} (S' | T') \]

\[ S \xrightarrow{\alpha} S' \]

\[ T \xrightarrow{\alpha} T' \]

\[ |\alpha=\alpha|T \xrightarrow{\alpha} T' \]

\[ |\alpha\neq\beta|T \xrightarrow{\alpha} T' \]

\[ !\alpha(x).T \xrightarrow{\alpha} T(\beta/x) !|\alpha(x).T. \]

All symmetric rules are omitted. In the third rule \(p \notin \lambda\) means that p does not appear in \(\lambda\). In the transition \(T \xrightarrow{(p)} T'\) the action \(\pi(p)\) is a bound output action. A π-process is a π-term in which all channel variables are bounded by input prefix operators and all private channels are bounded by scope operators. We write L, M, N, O, P, Q for processes. We write \(\lambda_a\) for an input or output action at channel a and \(\lambda_a\) for the complementary action. We will find it necessary to use the internal choice \(\tau.S + \tau.T\) defined by \(\{p\}(p.S | p.T)\).

The above semantics is called early semantics in literature. There is also a late semantics in which instantiation comes after the commitment of an input action [16]. If we think of bisimulation as a computational property, only early semantics makes sense. However it must be said that, due to the ever presence of name extrusion in π, the late semantics is intrinsic to the π-calculus [27]. We explain the claim by an example. The numbers fetchable at channel α can be defined in π as follows.

\[ [0]_\alpha \overset{\text{def}}{=} \pi(o).\pi(p).\pi(q).\overline{q}, \]

\[ [k+1]_\alpha \overset{\text{def}}{=} \pi(o).\pi(p).\pi(q).\overline{p}(r).\[k\]_\alpha. \]

It is simple to define the process \(a(x).if x=0 then P else Q\) that turns into \(P\) if the process has fetched a zero at channel a and turns into \(Q\) if the number it imports is not zero. Now
consider \( N \overset{\text{def}}{=} a(x).\ (if \ x=0 \ then \overline{e}x \ else \overline{d}x) \), \( C \overset{\text{def}}{=} !a(x).\overline{e}x \) and \( D \overset{\text{def}}{=} !a(x).\overline{d}x \). It is easy to see that the equality
\[
C | D | N = C | D
\]
fails for the \( \pi \)-calculus. The observer \( O \) defined in (2) can make a choice between 0 and 1 after it is engaged.
\[
O \overset{\text{def}}{=} a. b | \overline{p}(o). \overline{q}(o) \overline{r}(o). [0]_r.
\]
Let \( C | D | N | O \overset{\lambda}{\rightarrow} C | D | (o)(N' | O') \) be caused by the interaction between \( N \) and \( O \). This internal action cannot be bisimulated vacuously by \( C | D | O \) because \( C | D | O \) is observable at \( b \) whereas \( C | D | (o)(N' | O') \) is not. It cannot be bisimulated by \( C | D | (o)(C' | D | O' \) caused by the interaction between \( C \) and \( O \) since \( (o)(C' | D | O' \) is not observable at \( d \). Similarly the internal action of \( C | D | (o)(N' | O') \) caused by the interaction between \( D \) and \( O \) does not bisimulate \( C | D | N | O \overset{\lambda}{\rightarrow} C | D | (o)(N' | O') \). It is worth pointing out that (1) fails also for weak bisimilarity.

A lot can happen between the time a \( \pi \)-process starts to fetch a message and the time it sees the full picture of the message. This is the fundamental weakness of name-passing communications. In other type of model there are more efficient implementations of \( a(x).\ (if \ x=0 \ then \ P \ else \ Q \) that render (1) true. This simple example provides all the intuition for the main result of the paper.

The absolute equality for \( \pi \)-calculus can be equivalently defined in terms of input-output behaviours. Let’s call a \( \pi \)-term a quasi \( \pi \)-process if it does not contain any free channel variables.

\begin{definition}
A codivergent bisimulation \( R \) on quasi \( \pi \)-processes is an extensional bisimulation if the following statements are valid for all \( \lambda \neq \tau \) whenever \( SRT \):
\begin{enumerate}
\item If \( S \overset{\lambda}{\rightarrow} S' \), then \( T \overset{R}{\rightarrow} T'' \overset{\lambda}{\rightarrow} T' \) for some \( T'' \) and \( T' \) such that \( SRT'' \) and \( S'R'T' \).
\item If \( T \overset{\lambda}{\rightarrow} T' \), then \( S \overset{\lambda}{\rightarrow} S'' \overset{R}{\rightarrow} S' \) for some \( S'' \) and \( S' \) such that \( S''RT \) and \( S'R'T' \).
\end{enumerate}

The extensional bisimilarity \( \simeq_\pi \) is the largest extensional bisimulation.
\end{definition}

Clause (1) and clause (2) of the above definition ensure that equal processes have the same input-output behaviours. The next lemma points out the authoritative role of the absolute equality. It is the minimal equality for interactive objects that subsumes the extensional equality of computation.

\begin{lemma}
Suppose \( P, Q \) are \( \pi \)-processes. Then \( P =_\pi Q \) if and only if \( P \simeq_\pi Q \).
\end{lemma}

\begin{proof}
The relation \( \simeq_\pi \) is closed under concurrent composition and scope operation. Hence \( \simeq_\pi \subseteq =_\pi \). Conversely we show that the relation
\[
(S, T) \in \left\{ \begin{array}{l}
(S | T | p_1, \ldots, p_n | \overline{p_1} p_1 | \ldots | \overline{p_n} p_n | S) =_\pi (p_1, \ldots, p_n | \overline{p_1} p_1 | \ldots | \overline{p_n} p_n | T), \\
\text{ } p_1, \ldots, p_n \text{ are all the unbound private channels appearing in } S | T, \\
\text{ } \text{none of the global channels } a_1, \ldots, a_n \text{ appears in } S | T.
\end{array} \right\}
\]
is an extensional bisimulation. For details see [9].
\end{proof}

4 A General Negative Result

We argue in this section that roughly speaking if a model can release an infinite number of pairwise distinct complete messages, then the model cannot be interpreted in the \( \pi \)-calculus.
Suppose \( \mathcal{D} = \{[0]_a, [1]_a, [2]_a, \ldots, [i]_a, \ldots \} \) is an infinite set of processes in some model \( \mathcal{M} \) such that \( i \neq j \) implies
\[
[i]_a \neq [j]_a.
\] (3)

We assume that for every \( i \) the process \([i]_a\) can do one and only one action, which is an output action at channel \( a \), and turns into a process equivalent to \( 0 \) after the action. In other words,
\[
[i]_a \xrightarrow{\hat{i}} = 0,
\] (4)
where \( \hat{i} \) is the message released by \([i]_a\). There is no specific requirement on the messages \( 0, \hat{1}, \hat{2}, \ldots \) apart from that they are pairwise distinct. We require that in \( \mathcal{M} \) we can define an absorbing process \([\lambda x.0]_a\), a successor process \([\lambda x.x^+]_a\), a choice process \([\lambda x.\tau.x^++\tau]_a\), and a test process \([\lambda x.x > k]_a\) for each \( k \in \omega \). For each \( i \in \omega \) each of the processes can carry out one and only one action. Their operational behaviours are specified as follows:
\[
[\lambda x.0]_a \xrightarrow{a} = 0,
\] (5)
\[
[\lambda x.x^+]_a \xrightarrow{a} = [i+1]_a,
\] (6)
\[
[\lambda x.\tau.x^++\tau]_a \xrightarrow{a} = \tau.[i+1]_a + \tau,
\] (7)
\[
[\lambda x.x > k]_a \xrightarrow{a} = 0, \text{ if } i \leq k,
\] (8)
\[
[\lambda x.x > k]_a \xrightarrow{a} = [i+1]_a, \text{ if } i > k.
\] (9)

In \( \mathcal{M} \) there is also a replicated form for each of \([\lambda x.0]_a\), \([\lambda x.x^+]_a\) and \([\lambda x.\tau.x^++\tau]_a\), denoted respectively by \([!\lambda x.0]_a\), \([!\lambda x.x^+]_a\) and \([!\lambda x.\tau.x^++\tau]_a\). For each \( i \in \omega \) their unique actions are described as follows:
\[
[!\lambda x.0]_a \xrightarrow{\hat{i}} = ![\lambda x.0]_a,
\] (10)
\[
[!\lambda x.x^+]_a \xrightarrow{\hat{i}} = ![i+1]_a | ![\lambda x.x^+]_a,
\] (11)
\[
[!\lambda x.\tau.x^++\tau]_a \xrightarrow{\hat{i}} = (\tau.[i+1]_a + \tau) | ![\lambda x.\tau.x^++\tau]_a.
\] (12)

Let \( G \) be the process \([\lambda x.0]_a | ![\lambda x.x^+]_a | ![\lambda x.\tau.x^++\tau]_a \). We assume that in \( \mathcal{M} \) the following semantic equality is valid for each \( k \in \omega \):
\[
\Omega | G = \Omega | [\lambda x.x > k]_a.
\] (13)

Our assumption on \( \mathcal{M} \) is very liberal. It asks for no more than a numerical system that validates (13).

\textbf{Theorem 10}. \( \mathcal{M} \not\models \pi \).

\textbf{Proof}. Suppose there were a subbisimilarity \( \mathcal{Y} \) from the \( \mathcal{M} \)-processes to the \( \pi \)-processes. In what follows we write \( \mathcal{M}\pi \rightarrow \mathcal{M} \pi \rightarrow \) for \( \mathcal{M}\pi \rightarrow \) and \( \mathcal{P} \rightarrow \). Now fix \( a \in C_g \). We fix the notations for the interpretations of the \( \mathcal{M} \)-processes just described.

\begin{itemize}
  \item For each \( i \in \omega \) let \( N_i \) be the \( \pi \)-process such that \([i]_a \not\in \mathcal{N}_i \).
  \item Let \( C_0 \) be the \( \pi \)-process such that \([\lambda x.0]_a \not\in C_0 \).
  \item Let \( S_0 \) be the \( \pi \)-process such that \([\lambda x.x^+]_a \not\in S_0 \).
\end{itemize}
According to codivergence property there is no infinite computation sequence from \( N_2 \).

We now derive some properties for the \( Y \). Fu 22:7

This is because we also get an infinite sequence of computation by renaming and inserting scope operators it follows from (15) and extensionality that

\[
\Omega | C | S | E = \Omega | C | S | E | J_k. \tag{14}
\]

We now derive some properties for the \( \pi \)-processes \( N_0, N_1, N_2, N_3, \ldots \).

1. According to codivergence property there is no infinite computation sequence from \( S_0 | N_i \). Suppose \( S_0 | N_i \rightarrow B \). This action must be caused by \( S_0 \xrightarrow{\lambda a} S’ \) and \( N_i \xrightarrow{\sum} N’ \) for some complementary actions \( \lambda, \sum \) and some \( \pi \)-processes \( S’ \) and \( N’ \). If neither \( \lambda \) nor \( \lambda \) is a bound output action then \( S_0 | N_i \xrightarrow{\lambda a} \lambda a B \) and \( S_0 | N_i \xrightarrow{\lambda a} \lambda a B \).

It follows that \( B \rightarrow^* \lambda a \rightarrow^* \sum \rightarrow B’ = B \) and \( B \rightarrow^* \sum \rightarrow^* \lambda a \rightarrow B” = B \) for some \( B’, B” \). Now \( B | B \rightarrow^* B’ | B” = B | B \). We derive by induction that there would be an infinite computation sequence from \( S_0 | N_i | S_0 | N_i \), contradicting to the fact that \( [\lambda x.x^+ + \tau]_a \) is not convergent. If one of \( \lambda \) or \( \sum \) is a bound output action, we also get an infinite sequence of computation by renaming and inserting scope operators in appropriate places. More specifically without loss of generality suppose \( \sum = ao \) and \( \lambda a = p(o) \). Let \( S_0 \xrightarrow{ao} B_1 \) and \( N_i \xrightarrow{p(o)} B_2 \) for fresh private channels \( p, q \). Then

\[
B \equiv (o)(B_1\{o/p\} | B_2\{o/q\}). \tag{15}
\]

It follows from \( S_0 | N_i = B \) that there must exist some \( B’ \) such that

\[
S_0 | N_i \xrightarrow{ao} B_1 | B_2. \tag{16}
\]

is bisimulated by

\[
B \rightarrow^* \lambda a \rightarrow^* \pi(p) \rightarrow B’. \tag{17}
\]

Now (15) and (17) imply that (16) can be extended to

\[
S_0 | N_i \xrightarrow{ao} B_1 | B_2 \Rightarrow \lambda i \rightarrow^* B_1’ | B_2’. \tag{18}
\]

for some \( B_1’, B_2’ \) and some \( \lambda i, \lambda i \). Consequently the bisimulation (17) can be extended to

\[
B \rightarrow^* \lambda a \rightarrow^* \pi(p) \rightarrow B’ \Rightarrow \lambda i \rightarrow^* B’ | B”. \tag{19}
\]

for some \( B” \). The extension can be repeated infinitely often. We eventually get an infinite sequence with alternating input-output actions. Similarly we can derive from \( S_0 | N_i \xrightarrow{p(o)} ao B_1 | B_2 \) an infinite sequence with alternating output-input actions. In this way \( S_0 | N_i | S_0 | N_i \) would induce an infinite sequence of computation. This is again a contradiction. We conclude that \( S_0 | N_i \xrightarrow{-} N_{i+1} \) is essentially the only one-step nondeterministic computation of \( S_0 | N_i \).

2. \( N_i \) cannot do both an input action at channel \( a \) and an output action at channel \( a \). Otherwise \( N_i | N_i \) would be able to do an interaction, which would be a contradiction. This is because \( N_i | N_i \) cannot perform any nondeterministic computation since \( [i]_a | [i]_a \) cannot do any internal action. It cannot do a deterministic computation step since that would induce an infinite sequence of internal actions from \( N_i | N_i | N_i | N_i \), like in the previous case. So \( N_i \) may perform either an input action or an output action exclusively.
3. If $N_i$ can do an input, respectively output action then $N_j$ can do an input, respectively output action for all $j \in \omega$ since the latter has to interact with $C_0$.

4. It follows from $[\lambda x.0]_a \nleftarrow i = \emptyset$ that $C_0 \nrightarrow N_i \nleftarrow i = \emptyset$. Suppose $N_i \nrightarrow N'$ and $N_j \nrightarrow N''$ for some $N', N''$. Clearly $N' = \emptyset = N''$. But then one could derive from $N_i | S_0 \nrightarrow = N_{i+1}$ and $N_j | S_0 \nleftarrow = N_{j+1}$ that $N_{i+1} = N_{j+1}$, which implies $i = j$. We conclude that if both $N_i$ and $N_j$ can do free output actions at channel $a$ then the channels they release must be distinct whenever $i \neq j$.

5. According to our assumption we have $S_0 | N_i \nrightarrow = N_{i+1}$ for all $i \geq 0$. Let’s write $P \nrightarrow \lambda$ if $P \nrightarrow P'$ for some $P'$. It should be clear that if $N_1 \nrightarrow$, then $S_0 | N_0 \nrightarrow \nrightarrow$. Similarly if $N_2 \nrightarrow$, then $S_0 | N_1 \nrightarrow \nrightarrow$. Therefore $S_0 | S_0 \nrightarrow \nrightarrow$. By induction we can prove that if $N_i$ can release a global channel at $a$ then that global channel must appear in $S_0 | N_0$. So only a finite number of $N_0, N_1, N_2, \ldots$ can do free output actions. Let $h$ be the least number such that, for every $j \geq h$, $N_j$ does only a bound output action.

We prove the impossibility result by a case analysis on the actions of $N_0, N_1, N_2, N_3, \ldots$.

1. Suppose $k > h$. Let

$$\begin{align*}
N_h & \nrightarrow N'_h, \\
N_k & \nrightarrow N'_k.
\end{align*}$$

In this case $C_0, D_0, E_0, J_k, C, D$ and $E$ can only do input actions at channel $a$. Let

$$\begin{align*}
C_0 & \nrightarrow C'_0, \\
D_0 & \nrightarrow S'_0, \\
E_0 & \nrightarrow E'_0, \\
J_k & \nrightarrow J_p, \\
C & \nrightarrow C_p = C, \\
D & \nrightarrow S_p = S'_0 | S, \\
E & \nrightarrow E_p = E'_0 | E.
\end{align*}$$

None of $C_0, D_0, E_0, J_k, C, D$ and $E$ may perform any output actions. Otherwise there would be either an infinite deterministic computation or a nondeterministic computation step. It follows from (14) that $\Omega | C | S | E | J_k \nrightarrow \Omega | C | S | E | J_p$ must be bisimulated by one of the following.

$$\begin{align*}
\Omega | C | S | E & \nrightarrow \Omega | C_p | S | E, \\
\Omega | C | S | E & \nrightarrow \Omega | C | S_p | E, \\
\Omega | C | S | E & \nrightarrow \Omega | C | S | E_p.
\end{align*}$$

In the first case $\Omega | C | S | E | J_k \nrightarrow (p)(\Omega | C | S | E | J_p | N_k')$ should be bisimulated by $\Omega | C | S | E | N_k \nrightarrow (p)(\Omega | C_p | S | E | N_k')$ due to congruence. This is impossible because

$$\begin{align*}
(p)(\Omega | C | S | E | J_p | N_k') & \neq \Omega | C | S | E | N_{k+1} \\
& = (p)(\Omega | C | S | E | N_k') \\
& = (p)(\Omega | C_p | S | E | N_k')
\end{align*}$$

according to Lemma 7. In the second case $\Omega | C | S | E | J_k | N_h \nrightarrow (p)(\Omega | C | S | E | J_p | N_k')$ should be bisimulated by $\Omega | C | S | E | N_h \nrightarrow (p)(\Omega | C_p | S | E | N_k')$. Using again the full abstraction lemma one derives the following contradiction.
\[(p)(\Omega \mid C \mid S \mid E \mid J_p \mid N_k) = \Omega \mid C \mid S \mid E
\]

\[
\neq \Omega \mid C \mid S \mid E \mid N_{k+1}
\]

\[
= (p)(\Omega \mid C \mid S'_p \mid S \mid E \mid N'_k)
\]

\[
= (p)(\Omega \mid C \mid S_p \mid E \mid N'_k).
\]

In the third case \(\Omega \mid C \mid S \mid E \mid J_h \mid N_k \xrightarrow{\tau} (p)(\Omega \mid C \mid S \mid E \mid J_p \mid N_k)\) should be bisimulated by \(\Omega \mid C \mid S \mid E \mid N_k \xrightarrow{\tau} (p)(\Omega \mid C \mid S \mid E \mid N'_k)\). This is also impossible because \((p)(\Omega \mid C \mid S \mid E_p \mid N'_k) = (p)(\Omega \mid C \mid S \mid E \mid N'_k)\) can do a nondeterministic computation step, reaching to a state where \(N_{k+1}\) is not a concurrent component, while on the other hand \((p)(\Omega \mid C \mid S \mid E \mid J_p \mid N'_k) = \Omega \mid C \mid S \mid E \mid N_{k+1}\).

2. Now suppose the immediate actions of \(N_0, N_1, N_2, \ldots\) are input actions. In this case an output action of \(\Omega \mid C \mid S \mid E \mid J_h\) induced by \(J_h\), whether it is a free output action or a bound output action, must be bisimulated by an output action of \(\Omega \mid C \mid S \mid E\) induced by \(C\) or \(D\) or \(E\). We can deduce a contradiction as in the first case.

We have proved that there cannot be any subbisimilarity from \(\mathcal{M}\) to the \(\pi\)-calculus. ▶

5 Application to Higher Order Process Calculi

We demonstrate in this section that higher order process calculi cannot be interpreted in the \(\pi\)-calculus. Since the \(\pi\)-calculus is complete, it makes sense to focus on complete higher order process calculi. The completeness brings out the simplicity of the counter example and reveals the strength of the negative result.

Intuitively a model is complete if it is Turing powerful in an interactive fashion. A precise definition of completeness is given in terms of the \textit{computability model} denoted by \(\mathcal{C}\). The \(C\)-processes are generated from the following grammar.

\[P \ := \ 0 \mid \Omega \mid F^a_b(f(x)) \mid \pi(i) \mid P \mid P,\]

where \(f\) is a computable function and \(i \in \omega\). The semantics is defined by the following rules, in which \(f(i)\uparrow\) means that \(f\) is undefined on \(i\).

\[
\Omega \xrightarrow{\tau} \Omega \\
\pi(i) \xrightarrow{\pi(i)} 0 \\
F^a_b(f(x)) \xrightarrow{\pi(i)} b(j) \\
F^k_b(f(x)) \xrightarrow{\pi(i)} \Omega
\]

\[
P \xrightarrow{\pi(i)} P' \\
P \mid Q \xrightarrow{\pi(i)} P' \mid Q'
\]

The process \(\Omega\) diverges. The process \(F^a_b(f(x))\) simulates the input-output behaviour of \(f(x)\) with input channel \(a\) and output channel \(b\). The output process \(\pi(i)\) releases the number \(i\) at channel \(a\). The model \(\mathcal{C}\) is the minimal interactive extension of the computable functions. It says nothing about how computations are done. For this reason it is the best model to formalize the notion of completeness for interaction models. See [5] for more discussions and technical backgrounds.

Definition 11. An interaction model \(\mathcal{M}\) is \textit{complete} if \(\mathcal{C} \subseteq \mathcal{M}\).

The completeness of the \(\pi\)-calculus and the completeness of value-passing calculi are proved in [5]. In the same paper it is shown that there is no subbisimilarity from a value-passing calculus to \(\pi\). The reader should convince herself/himself that the value-passing processes satisfying (5) through (13) are easy to define.
We now convince the reader that the π-calculus cannot in general interpret higher order process calculi in a robust way. As an example we take a look at a particular functional model, denoted by \( I \). This is an abstraction-passing calculus. To define the model we need the syntactical class \( \mathcal{V} \) of abstraction variables. The elements of \( \mathcal{V} \) are denoted by \( U, V, W, X, Y, Z \). The set of \( \mathcal{I}-\text{terms} \) is generated by the following grammar:

\[
S, T := 0 | \mu(X).T | p[A].T | S | T | (p)T | A(\mu, \nu),
\]

\[
A := X | \lambda (u, v).T.
\]

An abstraction is either an abstraction variable or of the form \( \lambda (u, v).T \). The term \( A(\alpha, \beta) \) is an instantiation of \( A \) at \( \alpha, \beta \). We have the grammar equality \( \lambda (u, v).T(\alpha, \beta) \equiv A[\alpha/u, \beta/v] \). We abbreviate \( \lambda (u, v).T \) to \( \lambda (u).T \) if \( v \) does not appear in \( T \) and accordingly \( A(\alpha, \alpha) \) to \( A(\alpha) \). Because we would like to think that the abstraction \( A \) in \( p[A].T \) represents a complete message, we impose the following constraint.

(1) In \( p[A].T \) the abstraction \( A \) contains no free channel variables, no unbound private channels and no occurrences of global channel.

The constraint (1) gives rise to a simpler and weaker model whose algebraic property is easier to work out.

Using the action set \( \{ \alpha(A), p(A) \mid \alpha \in C_g \cup C_p \cup \{ \tau \} \} \), the simple operational semantics of \( I \) is defined by the following rules. Notice that because of the constraint (1) on abstractions there is no name extrusion.

\[
\begin{align*}
\alpha(X) & \vdash \lambda (A/X) & p & \not\rightarrow \lambda \\
\quad & | & & | & & | & & | \\
\mu(A) & \vdash \lambda (A) & (p)A & \not\rightarrow \lambda (A) \\
\quad & | & & | & & | & & | \\
\lambda (A) & \vdash \lambda (A) & \lambda (A) & \not\rightarrow \lambda (A) \\
\quad & | & & | & & | & & | \\
\end{align*}
\]

The forever diverging process can be defined as follows.

\[
\Omega \stackrel{\text{def}}{=} (p)(p(X) | (Xp | p[X]) | p[\lambda(X).x(X),(Xp | p[x]))).
\]

(20)

To demonstrate the power of \( I \) we provide a direct interpretation of the lazy \( \lambda \)-calculus \([1]\) in \( I \). Given an injective function from the set of \( \lambda \) variables to \( \mathcal{V} \), a \( \lambda \)-term \( M \) is interpreted as an abstraction by the following structural induction.

\[
\begin{align*}
[x] & = X, \\
\llbracket \lambda x.M \rrbracket & = \lambda (u, v).u(X).\llbracket M \rrbracket, \\
\llbracket MN \rrbracket & = \lambda (u, v).\langle mq \rangle \llbracket M \rrbracket \llbracket N \rrbracket .q(Z).Z(u, v).
\end{align*}
\]

The process \( [M](a, b) \) is a “function” that inputs a “\( \lambda \)-term” at channel \( a \) and output the result “\( \lambda \)-term” at channel \( b \). The structural aspect of the interpretation is enforced by the following simple lemma.

\textbf{Lemma 12.} \( [MN/X] \equiv [M][N/X] \).

Let \( =_{\beta} \) denote the \( \beta \)-conversion. The interpretation is semantically correct, guaranteed by the following facts. In the statement of Lemma 13 we identify \( 0 \mid P \) to \( P \) syntactically.

\textbf{Lemma 13.} For closed \( \lambda \)-terms \( M, N \), \( M \rightarrow N \) if and only if \( [M](a, b) \rightarrow [N](a, b) \).
Proof. The λ-term $M$ must be of the form $(\lambda x.L)M_1M_2\ldots M_k$, where the application is associative to the left. The encoding $\llbracket M \rrbracket(a, b)$ is by definition of the form 

$$(m_2q_2)((m_1q_1)((\lambda x.L)(m_1, q_1) | \overline{m_1}[\llbracket M_1 \rrbracket], q_1(Z), Z(m_2, q_2)) | \overline{m_2}[\llbracket M_2 \rrbracket], q_2(Z), Z(m_3, q_3))\ldots$$

which reduces in two deterministic computation steps to $L\{M_1/x\}M_2\ldots M_k\llbracket(a,b)\equiv\llbracket N \rrbracket(a, b)$ using Lemma 12. The argument is reversible. ◀

Lemma 14. For closed λ-terms $M, N$, $M \equiv N$ implies $\llbracket M \rrbracket(a, b) =_{L} \llbracket N \rrbracket(a, b)$.

Proof. In view of Lemma 13 this is the Church-Rosser property [2]. ◀

It is well-known [2] that there is a λ-term encoding $\llbracket 0 \rrbracket, \llbracket 1 \rrbracket, \ldots, \llbracket i \rrbracket, \ldots$ of the natural numbers and an encoding $\llbracket _- \rrbracket$ of the recursive functions such that $\llbracket f \rrbracket(\llbracket i \rrbracket) \rightarrow \llbracket f(i) \rrbracket$ if $f(i)$ is defined, and that $\llbracket f \rrbracket(\llbracket i \rrbracket) \rightarrow \rightarrow \ldots$ diverges if $f(i)$ is undefined. Given global channels $a, b$ we define the $L$-processes

$$\alpha(i) \overset{\text{def}}{=} \llbracket[\llbracket i \rrbracket] \rrbracket, \quad (21)$$

$$\lambda_{a,b}(f) \overset{\text{def}}{=} \llbracket[\llbracket f \rrbracket] \rrbracket(a, b). \quad (22)$$

It is straightforward to verify that if $f(i) = j$ then

$$\lambda_{a,b}(f) | \llbracket i \rrbracket \xrightarrow{a} \beta(j) \quad (23)$$

and if $f(i)$ is undefined then

$$\lambda_{a,b}(f) | \llbracket i \rrbracket \xrightarrow{a} \Omega. \quad (24)$$

The processes defined in (20), (21) and (22) actually provide an encoding of $C$ into $L$, hence the following.

Lemma 15. $C \subseteq L$.

Because of the constraint (‡) the completeness proof given in [32] cannot be carried out in $L$. The replication operator defined in [32] does not seem to be encodable in $L$.

To apply the general negative result it suffices to explain how numbers are defined in $L$. Given an abstraction $A$ we write $A^+$ for $\lambda(x, y).\overline{x}[A]$. Let

$$\hat{0} \overset{\text{def}}{=} \lambda(x, y).\overline{0}^+, \quad (\hat{k} + 1) \overset{\text{def}}{=} (\hat{k})^+.$$ 

The abstractions $\hat{0}, \hat{1}, \hat{2}, \ldots$ represent numbers. Using these numbers we can define the following simple processes.
\[ [i]_a \overset{\text{def}}{=} \pi[i], \]
\[ [\lambda x.0]_a \overset{\text{def}}{=} a(X), \]
\[ [\lambda x.x^+]_a \overset{\text{def}}{=} a(X) \cdot \pi[X^+], \]
\[ [\lambda x.\tau.x^++\tau]_a \overset{\text{def}}{=} a(X) \cdot (\tau \cdot \pi[X^+] + \tau), \]
\[ [\lambda x.x > k]_a \overset{\text{def}}{=} a(X) \cdot (n_1 a_1) \cdot (n_1 x_1) \cdot |n_1 X_1 \cdot n_2 a_2 \cdot (X_2 n_2 a_2) \cdot | \ldots | (n_k X_k \cdot n_k a_k) \cdot | n_k \cdot \pi[A^+] \ldots), \]
\[ [!\lambda x.0]_a \overset{\text{def}}{=} (p) \cdot (a(X) \cdot p(Z)) \cdot (Z(a,p) \cdot \pi[Z]) \cdot \pi(\lambda x.z) \cdot x(X) \cdot z(Z) \cdot (Z(x,z) \cdot \pi[Z])), \]
\[ [!\lambda x.x^+]_a \overset{\text{def}}{=} (p) \cdot (a(X) \cdot \pi[X^+] \cdot p(Z)) \cdot (Z(a,p) \cdot \pi[Z])) \]
\[ [!\lambda x.\tau.x^++\tau]_a \overset{\text{def}}{=} (p) \cdot (a(X) \cdot (\tau \cdot \pi[X^+] + \tau)) \cdot p(Z) \cdot (Z(a,p) \cdot \pi[Z])) \]
\[ \pi(\lambda x.z) \cdot x(X) \cdot ((\tau \cdot \pi[X^+] + \tau) \cdot z(Z) \cdot (Z(x,z) \cdot \pi[Z])), \]

The last three processes, \([!\lambda x.0]_a\), \([!\lambda x.x^+]_a\) and \([!\lambda x.\tau.x^++\tau]_a\), are intuitively the processes \(\lambda a(X)\), \(\lambda a(X) \cdot \pi[X^+]\) and \(\lambda a(X) \cdot (\tau \cdot \pi[X^+] + \tau)\). These replication processes must be implemented in L. It is easy to verify that the processes defined in the above satisfy the necessary requirements stated in (3) through (12).

Let \(G \overset{\text{def}}{=} [!\lambda x.0]_a \cdot [!\lambda x.x^+]_a \cdot [!\lambda x.\tau.x^++\tau]_a\). We now prove (13).

\section*{Lemma 16.} \(\Omega \mid G = \Omega \mid [\lambda x.x > k]_a\).

\textbf{Proof.} Consider the following transition

\[ \Omega \mid G \mid [\lambda x.x > k]_a \overset{a(A)}{\longrightarrow} \Omega \mid G \mid J_A, \tag{25} \]

where \(J_A\) is the following process

\(\begin{align*}
(n_1 a_1) \cdot (A(n_1, a_1)) \cdot (n_1 x_1) \cdot (n_2 a_2) \cdot (X_1 n_2 a_2) \cdot | \ldots | (X_k n_k a_k) \cdot | n_k \cdot \pi[A^+] \ldots)). \end{align*}\)

The process \(J_A\) and all its descendants can either carry out an internal action or enable \(\pi[A^+]\). They cannot do any other actions due to the constraint (1). Let \(J\) be the set of all terms \(J\) such that \(J_A \rightarrow J\). It can be partitioned into three disjoint subsets.

\[ \begin{align*}
J_0 & \overset{\text{def}}{=} \{ J' \in J \mid \text{no } \tau \text{ descendant of } J' \text{ can enable } \pi[A^+] \}, \\
J_1 & \overset{\text{def}}{=} \{ J' \in J \mid \text{every } \tau \text{ descendant of } J' \text{ can enable } \pi[A^+] \text{ now or in future} \}, \\
J_2 & \overset{\text{def}}{=} \overline{J_0 \cup J_1}. 
\end{align*} \]

If \(J_A \in J_0\) then (25) can be bisimulated by \(\Omega \mid G \overset{a(A)}{\longrightarrow} \Omega \mid G\) by invoking the component \([!\lambda x.0]_a\). Notice that \(J_A\) may induce an infinite computation. But this is not a problem in the presence of \(\Omega\). If \(J_A \in J_1\) then (25) can be bisimulated by \(\Omega \mid G \overset{a(A)}{\longrightarrow} \Omega \mid G \cdot \pi[A^+]\) by invoking the component \([!\lambda x.x^+]_a\). If \(J_A \in J_2\) then (25) can be bisimulated by \(\Omega \mid G \overset{a(A)}{\longrightarrow} \Omega \mid G \cdot (\tau \cdot \pi[A^+] + \tau)\) by invoking the component \([!\lambda x.\tau.x^++\tau]_a\). We only have to prove that every \(J' \in J_2\) renders true the following equality.

\[ \Omega \mid (\tau \cdot \pi[A^+] + \tau) = \Omega \mid J'. \tag{26} \]
Suppose \( J' \xrightarrow{\tau} J'' \). It induces the transition
\[
\Omega | J' \xrightarrow{\tau} \Omega | J''.
\] (27)

If \( J'' \in \mathcal{J}_1 \), then (27) is bisimulated by \( \Omega | (\tau.\pi[A^+\tau] + \tau) \xrightarrow{\tau} \Omega | (\tau.\pi[A^+\tau] + \tau) \). If \( J'' \in \mathcal{J}_0 \), then (27) is bisimulated by \( \Omega | (\tau.\pi[A^+\tau] + \tau) \xrightarrow{\tau} \Omega | \pi[A^+\tau] \). We are done.

The main result of the paper now follows.

Theorem 17. \( L \not\sqsubseteq \pi \).

Theorem 17 provides a seemingly contradictory result to the well-known encoding of higher order \( \pi \)-calculus \( \pi^\omega \) in the first order \( \pi \)-calculus [25, 26] and other similar encodings [29, 31]. The higher order process calculi studied in these encodings are both abstraction-passing and complete. The criteria for relative expressiveness adopted in these papers are weaker than the one of this paper. They include extensionality, equipollence and weak bisimulation. Some of the encodings satisfy full abstraction property, others satisfy only a weaker form of it. Almost all encodings \textit{op. cit.} satisfy codivergence and branching bisimulation property. As criteria for submodel relationship codivergence and branching bisimulation help to derive reasonable properties about interpretations, which is duly demonstrated in the proof of the general negative result. Without these two criteria we are not able to conclude for example that \( N_i \) cannot perform both an input action and an output action.

The source models considered in the above mentioned papers are more powerful than \( L \) in the sense that the latter is restricted by the condition (‡). The equality stated in (13) fails in \( \pi^\omega \) since an observer can be powerful enough to detect the presence of the component \([\lambda x.x > k]_\alpha\). Thus Theorem 17 does not contradict to the expressiveness results given by the encodings in the afore mentioned papers, at least on the face of it. We would however like to draw reader’s attention to two points. Firstly if we impose the restriction (‡) to \( \pi^\omega \) we get a variant, denoted by say \( \pi^\omega_0 \), that is comparable to \( L \). The proof of Theorem 17 applies to \( \pi^\omega_0 \). It follows that \( \pi^\omega_0 \not\sqsubseteq \pi \). On the other hand Sangiorgi’s encoding of \( \pi^\omega \) in \( \pi \) is also an encoding of \( \pi^\omega_0 \) in \( \pi \). We do not understand at the moment is if the encoding is still fully abstract. Secondly if we drop the condition (‡) on \( L \), we get a model denoted by \( H \). We may ask the question if \( L \not\subseteq H \). Notice that \( L \subseteq H \) implies \( H \not\subseteq \pi \). Notice also that the failure of (13) implies that the identity map from \( L \) to \( H \) is not a subbisimilarity; it does not rule out however that \( L \subseteq H \). For the same reason \( \pi^\omega_0 \subseteq \pi^\omega \) would imply \( \pi^\omega \subseteq \pi \). We do not expect that either of the questions, “\( L \subseteq H \)” and “\( \pi^\omega_0 \subseteq \pi^\omega \)?”, is easy to answer. A possible way to attack these problems is to explore the power of universal processes [6]. This is left for future investigation.

The negative result offered by Theorem 17 provides a fresh look at the issue of higher order calculi vs name-passing calculi and forces us to ask some deeper questions.

6 Conclusion

Name-passing calculi are low level models. It should not come as a surprise that neither the value-passing calculi nor the higher order process calculi with complete messages can be interpreted by the \( \pi \)-calculus. The size of a message in the former models is unbounded. In such models in a single interaction a number/abstraction of arbitrary size is passed from one process to another. This is a high level feature. Studies of communication mechanisms at different abstract levels are part of process theory.
Let’s summarize what we know about the relative expressiveness of the three classes of model. Now “Name-Passing $⊑$ Value-Passing $⊑$ Name-Passing” [5] and for what we know at the moment “Process-Passing $⊑$ Name-Passing”. It is easily seen that “Process-Passing $⊑$ Value-Passing” because value-passing communications cannot interpret name extrusion. The problem “Value-Passing $⊑$ Process-Passing?” deserves attention. The numbers can be coded up by abstractions. The question is how to do it in a bijective way.

Having seen all the negative results, one wonders if there is a communication mechanism that can interpret the name-passing, the value-passing and the abstraction-passing mechanisms. In [7] a universal model $V$ is defined that can indeed interpret for example the model $L$, the $\pi$-calculus and the value-passing calculus [3]. Such a model, rather than being artificial, is well motivated by the Church-Turing Thesis. It is against this overall picture that the result of this short paper is to be appreciated.

Acknowledgments. We thank the anonymous reviewers for their criticisms and comments. We thank Davide Sangiorgi for his expert advices. We thank BASICS members for their interest.

References


Abstract

Probabilistic automata (PA) combine probability and nondeterminism. They can be given different semantics, like strong bisimilarity, convex bisimilarity, or (more recently) distribution bisimilarity. The latter is based on the view of PA as transformers of probability distributions, also called belief states, and promotes distributions to first-class citizens.

We give a coalgebraic account of the latter semantics, and explain the genesis of the belief-state transformer from a PA. To do so, we make explicit the convex algebraic structure present in PA and identify belief-state transformers as transition systems with state space that carries a convex algebra. As a consequence of our abstract approach, we can give a sound proof technique which we call bisimulation up-to convex hull.

1998 ACM Subject Classification F.3 Logics and Meanings of Programs, G.3 Probability and Statistics, F.1.2 Modes of Computation, D.2.4 Software/Program verification

Keywords and phrases belief-state transformers, bisimulation up-to, coalgebra, convex algebra, convex powerset monad, probabilistic automata

1 Introduction

Probabilistic automata (PA), closely related to Markov decision processes (MDPs), have been used along the years in various areas of verification [40, 37, 38, 2], machine learning [24, 41], and semantics [66, 52]. Recent interest in research around semantics of probabilistic programming languages has led to new insights in connections between category theory, probability theory, and automata [59, 12, 27, 58, 44].

PA have been given various semantics, starting from strong bisimilarity [39], probabilistic (convex) bisimilarity [50, 49], to bisimilarity on distributions [18, 14, 10, 21, 11, 25, 22, 26]. In this last view, probabilistic automata are understood as transformers of belief states, labeled transition systems (LTSs) having as states probability distributions, see e.g. [14, 15, 35, 1, 13, 22, 19]. Checking such equivalence raises a lot of challenges since belief-states are uncountable. Nevertheless, it is decidable [26, 20] with help of convexity. Despite these developments, what remains open is the understanding of the genesis of belief-state transformers and canonicity of distribution bisimilarity, as well as the role of convex algebras.
The theory of coalgebras [30, 46, 31] provides a toolbox for modelling and analysing different types of state machines. In a nutshell, a coalgebra is an arrow \( c: S \to FS \) for some functor \( F: C \to C \) on a category \( C \). Intuitively, \( S \) represents the space of states of the machine, \( c \) its transition structure and the functor \( F \) its type. Most importantly, every functor gives rise to a canonical notion of behavioural equivalence \( (\approx) \), a coinductive proof technique and, for finite states machines, a procedure to check \( \approx \).

By tuning the parameters \( C \) and \( F \), one can retrieve many existing types of machines and their associated equivalences. For instance, by taking \( C = \text{Sets} \), the category of sets and functions, and \( FS = (\mathbb{P}DS)^L \), the set of functions from \( L \) to subsets \( (\mathbb{P}) \) of probability distributions \( D \) over \( S \), coalgebras \( c: S \to FS \) are in one-to-one correspondence with PA with labels in \( L \). Moreover, the associated notion of behavioural equivalence turns out to be the classical strong probabilistic bisimilarity of [39] (see [4, 54] for more details). Recent work [43] shows that, by taking a slightly different functor, forcing the subsets to be convex, one obtains probabilistic (convex) bisimilarity as in [50, 49].

In this paper, we take a coalgebraic outlook at the semantics of probabilistic automata as belief-state transformers: we wish to translate a PA \( M \) into a labeled transition system on convex algebras. In turn, these can be transformed – simply forgetting the algebraic structure – into standard LTSs on convex algebras. This construction requires the existence of a lifting \( F \) of the category of algebras for the monad \( M \). In the case of PA, one lifts the states space to \( \mathbb{P}S \), the set of functions from \( S \) to \( \mathbb{P}S \), which is a coalgebra for the functor \( FX = 2 \times (\mathbb{P}S)^L \). The result of the determinisation is a deterministic automaton \( c^\#: \mathbb{P}S \to 2 \times (\mathbb{P}S)^L \) (with state space \( \mathbb{P}S \)), which is a coalgebra for the functor \( FX = 2 \times X^L \). In the case of PA, one lifts the states space to \( D \mathbb{S} \), in the one of NDA to \( \mathbb{P}S \). From an abstract perspective, both \( D \) and \( \mathbb{P} \) are monads, hereafter denoted by \( \mathcal{M} \), and both PA and NDA can be regarded as coalgebras of type \( c: S \to F\mathcal{M}S \).

In [53], a generalised determinisation transforming coalgebras \( c: S \to F\mathcal{M}S \) into coalgebras \( c^\#: MS \to F\mathcal{M}S \) was presented. This construction requires the existence of a lifting \( F \) of \( \mathcal{M} \) to the category of algebras for the monad \( \mathcal{M} \). In the case of NDA, the functor \( FX = 2 \times X^L \) can be easily lifted to the category of join-semilattices (algebras for \( \mathbb{P} \)) and, the coalgebra \( c^\#: \mathbb{P}S \to 2 \times (\mathbb{P}S)^L \) resulting from this construction turns out to be exactly the standard determinised automaton. Unfortunately, this is not the case with probabilistic automata: because of the lack of a suitable distributive law of \( D \) over \( \mathbb{P} \) [64], it is impossible to suitably lift \( FX = (\mathbb{P}X)^L \) to the category of convex algebras (algebras for the monad \( D \)).

The way out of the impasse consists in defining a powerset-like functor on the category of convex algebras. This is not a lifting but it enjoys enough properties that allow to lift every PA into a labeled transition system on convex algebras. In turn, these can be transformed – without changing the underlying behavioural equivalence – into standard LTSs on \( \text{Sets} \) by simply forgetting the algebraic structure. We show that the result of the whole procedure is exactly the expected belief-state transformer and that the induced notion of behavioural equivalence coincides with a canonical one present in the literature [14, 25, 22, 26].

The analogy with NDA pays back in terms of proof techniques. In [6], Bonchi and Pous introduced an efficient algorithm to check language equivalence of NDA based on coinduction up-to [45]: in a determinised automaton \( c^\#: \mathbb{P}S \to 2 \times (\mathbb{P}S)^L \), language equivalence can be proved by means of bisimulations up-to the structure of join semilattice carried by the state space \( \mathbb{P}S \). Algorithmically, this results in an impressive pruning of the search space.

Similarly, in a belief-state transformer \( c^\#: DS \to (\mathbb{P}DS)^L \), one can coinductively reason up-to the convex algebraic structure carried by \( DS \). The resulting proof technique, which we call in this paper bisimulation up-to convex hull, allows finite relations to witness the equivalence of infinitely many states. More precisely, by exploiting a recent result in convex
3.1. The incidence of the abstract and concrete transformers and semantics. We present bisimulation the transformation from PA to belief-state transformers in Section 6 and prove the coalgebra (Section 4). We provide the PA functor on convex algebras in Section 5. We give (strong probabilistic) bisimulation for the rest of the section, we fix a PA \( M = (S, L, \rightarrow) \).

\[ s \xrightarrow{a} \xi \Rightarrow \exists \xi' \in \mathcal{D}(S). t \xrightarrow{a} \xi' \land \xi \equiv_R \xi', \quad \text{and} \quad t \xrightarrow{a} \xi' \Rightarrow \exists \xi \in \mathcal{D}(S). s \xrightarrow{a} \xi \land \xi \equiv_R \xi'. \]

Here, \( \equiv_R \subseteq \mathcal{D}(S) \times \mathcal{D}(S) \) is the lifting of \( R \) to distributions, defined by \( \xi \equiv_R \xi' \) if and only if there exists a distribution \( \nu \in \mathcal{D}(S \times S) \) such that

1. \( \sum_{s \in S} \nu(s, t) = \xi(s) \) for any \( s \in S \),
2. \( \sum_{s \in S} \nu(s, t) = \xi'(t) \) for any \( t \in T \), and
3. \( \nu(s, t) \neq 0 \) implies \( (s, t) \notin R \).

Two states \( s \) and \( t \) are (strongly probabilistically) bisimilar, notation \( s \sim t \), if there exists a (strong probabilistic) bisimulation \( R \) with \( (s, t) \in R \).
Definition 3 (Convex Bisimilarity). A relation \( R \subseteq S \times S \) is a convex (probabilistic) bisimulation if \( (s, t) \in R \) implies, for all actions \( a \in L \) and all \( \xi \in \mathcal{D}(S) \), that
\[
s \overset{a}{\to} \xi \Rightarrow \exists \xi' \in \mathcal{D}(S). t \overset{a}{\to} \xi' \land \xi \equiv_R \xi', \quad \text{and} \quad t \overset{a}{\to} \xi' \Rightarrow \exists \xi \in \mathcal{D}(S). s \overset{a}{\to} \xi \land \xi \equiv_R \xi'.
\]

Here \( \to_c \) denotes the convex transition relation, defined as follows: \( s \overset{a}{\to}_c \xi \) if and only if \( \xi = \sum_{i=1}^{n} p_i \xi_i \) for some \( \xi_i \in \mathcal{D}(S) \) and \( p_i \in [0, 1] \) satisfying \( \sum_{i=1}^{n} p_i = 1 \) and \( s \overset{a}{\to} \xi_i \) for \( i = 1, \ldots, n \). Two states \( s \) and \( t \) are convex bisimilar, notation \( s \sim_c t \), if there exists a convex bisimulation \( R \) with \( (s, t) \in R \).  

Convex bisimilarity is (strong probabilistic) bisimilarity on the "convex closure" of the given PA. More precisely, consider the PA \( M_c = (S, L, \to_c) \) in which \( s \overset{a}{\to}_c \xi \) whenever \( s \in S \) and \( \xi \) is in the convex hull (see Section 3 for a definition) of the set \( \{ \xi \in \mathcal{D}(S) \mid s \overset{a}{\to} \xi \} \). Then convex bisimilarity of \( M \) is bisimilarity of \( M_c \). Hence, if bisimilarity is the behavioural equivalence of interest, we see that convex semantics arises from a different perspective on the representation of a PA: instead of seeing the given transitions as independent, we look at them as generators of infinitely many transitions in the convex closure.

There is yet another way to understand PA, as belief-state transformers, present but sometimes implicit in [10, 25, 22, 13, 15, 14, 26, 11] to name a few, with behavioural equivalences on distributions. We were particularly inspired by the original work of Deng et al. [13, 15, 14] as well as [26]. Given a PA \( M = (S, L, \to) \), consider the labeled transition system \( M_{bs} = (DS, L, \overset{\to}{\to}) \) with states distributions over the original states of \( M \), and transitions \( \overset{\to}{\to} \subseteq DS \times L \times DS \) defined by
\[
\xi \overset{a}{\to} \zeta \iff \xi = \sum p_i s_i, \; s_i \overset{a}{\to} \xi_i, \; \xi = \sum p_i \xi_i.
\]
We call \( M_{bs} \) the belief-state transformer of \( M \). Figure 1, right, displays a part of the belief-state transformer induced by the PA of Figure 1, left. According to this definition, a distribution makes an action step only if all its support states can make the step.

This, and hence the corresponding notion of bisimulation, can vary. For example, in [26] a distribution makes a transition \( \overset{a}{\to} \) if some of its support states can perform an \( \overset{a}{\to} \) step. There are several proposed notions of equivalences on distributions [25, 18, 19, 22, 13, 10, 26] that mainly differ in the treatment of termination. See [26] for a detailed comparison.

Definition 4 (Distribution Bisimilarity). An equivalence \( R \subseteq DS \times DS \) is a distribution bisimilarity of \( M \) if and only if it is a bisimulation of the belief-state transformer \( M_{bs} \).

Two distributions \( \xi \) and \( \zeta \) are distribution bisimilar, notation \( \xi \sim_d \zeta \), if there exists a bisimulation \( R \) with \( (\xi, \zeta) \in R \). Two states \( s \) and \( t \) are distribution bisimilar, notation \( s \sim_d t \), if \( \delta_s \sim_d \delta_t \), where \( \delta_s \) denotes the Dirac distribution with \( \delta_s(x) = 1 \).  

While the foundations of strong probabilistic bisimilarity are well-studied [54, 4, 65] and convex probabilistic bisimilarity was also recently captured coalgebraically [43], the foundations of the semantics of PA as transformers of belief states is not yet explained. One of the goals of the present paper is to show that also that semantics (naturally on distributions [26]) is an instance of generic behavioural equivalence. Note that a (somewhat concrete) proof is given for the bisimilarity of [26] — the authors have proven that their bisimilarity is coalgebraic bisimilarity of a certain coalgebra corresponding to the belief-state transformer. What is missing there, and in all related work, is an explanation of the relationship of the belief-state transformer to the original PA. Clarifying the foundations of the belief-state transformer and the distribution bisimilarity is our initial motivation.
3 Convex Algebras

By $\mathcal{C}$ we denote the signature of convex algebras

$$\mathcal{C} = \{ (p_i)_{i=0}^n \mid n \in \mathbb{N}, p_i \in [0,1], \sum_{i=0}^n p_i = 1 \}. $$

The operation symbol $(p_i)_{i=0}^n$ has arity $(n+1)$ and it will be interpreted by a convex combination with coefficients $p_i$ for $i = 0, \ldots, n$. For $p \in [0,1]$ we write $\bar{p} = 1 - p$.

Definition 5. A convex algebra $X$ is an algebra with signature $\mathcal{C}$, i.e., a set $X$ together with an operation $\sum_{i=0}^n p_i(-)$, for each operational symbol $(p_i)_{i=0}^n \in \mathcal{C}$, such that the following two axioms hold:
- Projection: $\sum_{i=0}^n p_i x_i = x_j$ if $p_j = 1$.
- Barycenter: $\sum_{i=0}^n p_i \left( \sum_{j=0}^m q_i x_j \right) = \sum_{j=0}^m (\sum_{i=0}^n p_i q_i) x_j$.

A convex algebra homomorphism $h$ from $X$ to $Y$ is a convex (synonymously, affine) map, i.e., $h: X \to Y$ with the property $h(\sum_{i=0}^n p_i x_i) = \sum_{i=0}^n p_i h(x_i)$. \hfill \lozenge

Remark 6. Let $X$ be a convex algebra. Then (for $p_n \neq 1$)

$$\sum_{i=0}^n p_i x_i = \frac{\sum_{j=0}^{n-1} \bar{p_j} x_j}{\sum_{j=0}^n \bar{p_j}} + p_n x_n \quad (1)$$

Hence, an $(n+1)$-ary convex combination can be written as a binary convex combination using an $n$-ary convex combination. As a consequence, if $X$ is a set that carries two convex algebras $X_1$ and $X_2$ with operations $\sum_{i=0}^n p_i(-)$ and $\bigoplus_{i=0}^n p_i(-)$, respectively (and binary versions $+$ and $\oplus$, respectively) such that $px + \bar{p}y = px \oplus \bar{p}y$ for all $p, x, y$, then $X_1 = X_2$.

One can also see (1) as a definition, see e.g. [60, Definition 1]. We make the connection explicit with the next proposition, cf. [60, Lemma 1-Lemma 4].

Proposition 7. Let $X$ be a set with binary operations $px + \bar{p}y$ for $x, y \in X$ and $p \in (0,1)$. For $x, y, z \in X$ and $p, q \in (0,1)$, assume
- Idempotence: $px + \bar{p}x = x$,
- Parametric commutativity: $px + \bar{p}y = \bar{p}y + px$,
- Parametric associativity: $p(qx + \bar{p}y) + \bar{p}z = pqx + \bar{p}q \left( \frac{p}{\bar{p}} y + \frac{\bar{p}}{p} z \right)$,

and define $n$-ary convex operations by the projection axiom and the formula (1). Then $X$ becomes a convex algebra. \hfill \blacktriangleleft

Hence, it suffices to consider binary convex combinations only, whenever more convenient.

Definition 8. Let $X$ be a convex algebra, with carrier $X$ and $C \subseteq X$. $C$ is convex if it is the carrier of a subalgebra of $X$, i.e., if $px + \bar{p}y \in C$ for $x, y \in C$ and $p \in (0,1)$. The convex hull of a set $S \subseteq X$, denoted $\text{conv}(S)$, is the smallest convex set that contains $S$. \hfill \lozenge

Clearly, a set $C \subseteq X$ for $X$ being the carrier of a convex algebra $X$ is convex if and only if $C = \text{conv}(C)$. Convexity plays an important role in the semantics of probabilistic automata, for example in the definition of convex bisimulation, Definition 3.

\footnote{Stone’s cancellation Postulate V is not used in his Lemma 1-Lemma 4.}
4 Coalgebras

In this section, we briefly review some notions from (co)algebra which we will use in the rest of the paper. This section is written for a reader familiar with basic category theory. We have included an expanded version of this section in the full version that also includes basic categorical definitions and more details than what we do here.

Coalgebras provide an abstract framework for state-based systems. Let $C$ be a base category. A coalgebra is a pair $(S, c)$ of a state space $S$ (object in $C$) and an arrow $c : S \rightarrow FS$ in $C$ where $F : C \rightarrow C$ is a functor that specifies the type of transitions. We will sometimes just say the coalgebra $(S, c)$. A coalgebra homomorphism from a coalgebra $(S, c)$ to a coalgebra $(T, d)$ is an arrow $h : S \rightarrow T$ in $C$ that makes the diagram on the right commute.

Coalgebras of a functor $F$ and their coalgebra homomorphisms form a category that we denote by $\text{Coalg}_C(F)$. Examples of functors on $\text{Sets}$ which are of interest to us are:

1. The constant exponent functor $(-)^L$ for a set $L$, mapping a set $X$ to the set $X^L$ of all functions from $L$ to $X$, and a function $f : X \rightarrow Y$ to $f^L : X^L \rightarrow Y^L$ with $f^L(y) = f \circ g$.

2. The powerset functor $\mathcal{P}$ mapping a set $X$ to its powerset $\mathcal{P}X = \{S \mid S \subseteq X\}$ and on functions $f : X \rightarrow Y$ given by direct image: $\mathcal{P}f : \mathcal{P}X \rightarrow \mathcal{P}Y$, $\mathcal{P}(f)(U) = \{f(u) \mid u \in U\}$.

3. The finitely supported probability distribution functor $\mathcal{D}$ is defined, for a set $X$ and a function $f : X \rightarrow Y$, as
   \[ \mathcal{D}X = \{ \varphi : X \rightarrow [0, 1] \mid \sum_{x \in X} \varphi(x) = 1, \text{supp}(\varphi) \text{ is finite} \}, \]
   \[ \mathcal{D}f(\varphi)(y) = \sum_{x \in f^{-1}(y)} \varphi(x). \]

   The support set of a distribution $\varphi \in \mathcal{D}X$ is defined as $\text{supp}(\varphi) = \{x \in X \mid \varphi(x) \neq 0\}$.

4. The functor $C$ $[43, 28, 63]$ maps a set $X$ to the set of all nonempty convex subsets of $X$, and a function $f : X \rightarrow Y$ to the function $\mathcal{P}f$.

We will often decompose $\mathcal{P}$ as $\mathcal{P}_{ne} + 1$ where $\mathcal{P}_{ne}$ is the nonempty powerset functor and $(-) + 1$ is the termination functor defined for every set $X$ by $X + 1 = X \cup \{\ast\}$ with $\ast \notin X$ and every function $f : X \rightarrow Y$ by $f + 1(\ast) = \ast$ and $f + 1(x) = x$ for $x \in X$.

Coalgebras over a concrete category are equipped with a generic behavioural equivalence, which we define next. Let $(S, c)$ be an $F$-coalgebra on a concrete category $C$, with $\mathcal{U} : C \rightarrow \text{Sets}$ being the forgetful functor. An equivalence relation $R \subseteq \mathcal{U}S \times \mathcal{U}S$ is a kernel bisimulation (synonymously, a cocongruence) $[57, 36, 67]$ if it is the kernel of a homomorphism, i.e., $R = \ker \mathcal{U}h = \{(s, t) \in \mathcal{U}S \times \mathcal{U}S \mid \mathcal{U}h(s) = \mathcal{U}h(t)\}$ for some coalgebra homomorphism $h : (S, c) \rightarrow (T, d)$ to some $F$-coalgebra $(T, d)$. Two states $s, t$ of a coalgebra are behaviourally equivalent notation $s \approx t$ iff there is a kernel bisimulation $R$ with $(s, t) \in R$. A simple but important property is that if there is a functor from one category of coalgebras (over a concrete category) to another that preserves the state space and is identity on morphisms, then this functor preserves behavioural equivalence: if two states are equivalent in a coalgebra of the first category, then they are also equivalent in the image under the functor in the second category.

We are now in position to connect probabilistic automata to coalgebras.

Proposition 9 $[4, 54]$. A probabilistic automaton $M = (S, L, \rightarrow)$ can be identified with a $(\mathcal{PD})^L$-coalgebra $c_M : S \rightarrow (\mathcal{PD}S)^L$ on $\text{Sets}$, where $s \xrightarrow{\xi} s'$ in $M$ iff $\xi \in c_M(s)(a)$ in $(S, c_M)$. Bisimilarity in $M$ equals behavioural equivalence in $(S, c_M)$, i.e., for two states $s, t \in S$ we have $s \sim t \iff s \approx t$.

It is also possible to provide convex bisimilarity semantics to probabilistic automata via coalgebraic behavioural equivalence, as the next proposition shows.
**Proposition 10** ([43]). Let \( M = (S, L, \to) \) be a probabilistic automaton, and let \((S, \bar{c}_M)\) be a \((C + 1)^L\)-coalgebra on \( \text{Sets} \) defined by \( \bar{c}_M(s)(a) = \text{conv}(c_M(s)(a)) \) where \( c_M \) is as before, if \( c_M(s)(a) = \{ \xi \mid s \xrightarrow{} \xi \} \neq \emptyset \); and \( \bar{c}_M(s)(a) = \ast \) if \( c_M(s)(a) = \emptyset \). Convex bisimilarity in \( M \) equals behavioural equivalence in \((S, \bar{c}_M)\).

The connection between \((S, c_M)\) and \((S, \bar{c}_M)\) in Proposition 10 is the same as the connection between \( M \) and \( M_c \) in Section 2. Abstractly, it can be explained using the following well known generic property.

**Lemma 11** ([46, 4]). Let \( \sigma : F \Rightarrow G \) be a natural transformation from \( F : C \to C \) to \( G : C \to C \). Then \( T : \text{Coalg}_{C}(F) \to \text{Coalg}_{C}(G) \) given by \( T(S, \xi) = (S, \xi) \) on objects and identity on morphisms is a functor that preserves behavioural equivalence. If \( \sigma \) is injective, then \( T \) also reflects behavioural equivalence.

**Example 12.** We have that \( \text{conv} : \mathcal{PD} \to C + 1 \) given by \( \text{conv}(\emptyset) = \ast \) and \( \text{conv}(X) \) is the already-introduced convex hull for \( X \subseteq DS, X \neq \emptyset \) is a natural transformation. Therefore, \( \text{conv}^L : (\mathcal{PD})^L \Rightarrow (C + 1)^L \) is one as well, defined pointwise. As a consequence from Lemma 11, we get a functor \( T_{\text{conv}} : \text{Coalg}_{\text{Sets}}((\mathcal{PD})^L) \to \text{Coalg}_{\text{Sets}}((C + 1)^L) \) and hence bisimilarity implies convex bisimilarity in probabilistic automata.

Also, an injective natural transformation \( \iota : C + 1 \Rightarrow \mathcal{PD} \) is given by \( \iota(X) = X \) and \( \iota(\ast) = \emptyset \) yielding an injective \( \chi : (C + 1)^L \Rightarrow (\mathcal{PD})^L \). As a consequence, convex bisimilarity coincides with strong bisimilarity on the “convex-closed” probabilistic automaton \( M_c \), i.e., the coalgebra \((S, \bar{c}_M)\) whose transitions are all convex combinations of \( M \)-transitions.

### 4.1 Algebras for a Monad

The behaviour functor \( F \) often is, or involves, a monad \( \mathcal{M} \), providing certain computational effects, such as partial, non-deterministic, or probabilistic computations.

More precisely, a monad is a functor \( \mathcal{M} : \text{C} \to \text{C} \) together with two natural transformations: a unit \( \eta : \text{id}_\text{C} \Rightarrow \mathcal{M} \) and multiplication \( \mu : \mathcal{M}^2 \Rightarrow \mathcal{M} \) that satisfy the laws

\[
\mu \circ \eta \circ \mathcal{M} = \text{id} = \mu \circ \mathcal{M} \eta \text{ and } \mu \circ \mu \mathcal{M} = \mu \circ \mathcal{M} \mu.
\]

An example that will be pivotal for our exposition is the finitely supported distribution monad. The unit of \( \mathcal{D} \) is given by a Dirac distribution \( \eta(x) = \delta_x \) for \( x \in X \) and the multiplication by \( \mu(\Phi)(x) = \sum_{\varphi \in \text{supp}(\Phi)} \Phi(\varphi) \cdot \varphi(x) \) for \( \Phi \in \mathcal{D} \mathcal{D}X \).

With a monad \( \mathcal{M} \) on a category \( \text{C} \) one associates the Eilenberg-Moore category \( \text{EM}(\mathcal{M}) \) of Eilenberg-Moore algebras. Objects of \( \text{EM}(\mathcal{M}) \) are pairs \( A = (A, a) \) of an object \( A \in \text{C} \) and an arrow \( a : \mathcal{M}A \to A \) satisfying \( a \circ \eta_A = \text{id} \) and \( a \circ \mathcal{M}a = a \circ \mu \).

A homomorphism from an algebra \( A = (A, a) \) to an algebra \( B = (B, b) \) is a map \( h : A \to B \) in \( \text{C} \) between the underlying objects satisfying \( h \circ a = b \circ \mathcal{M}h \).

A category of Eilenberg-Moore algebras which is particularly relevant for our exposition is described in the following proposition. See [61] and [51] for the original result, but also [16, 17] or [29, Theorem 4] where a concrete and simple proof is given.

**Proposition 13** ([61, 16, 17, 29]). Eilenberg-Moore algebras of the finitely supported distribution monad \( \mathcal{D} \) are exactly convex algebras as defined in Section 3. The arrows in the Eilenberg-Moore category \( \text{EM}(\mathcal{D}) \) are convex algebra homomorphisms.

As a consequence, we will interchangeably use the abstract (Eilenberg-Moore algebra) and the concrete definition (convex algebra), whatever is more convenient. For the latter, we also just use binary convex operations, by Proposition 7, whenever more convenient.
The Power of Convex Algebras

4.2 The Generalised Determinisation

We now recall a construction from [53], which serves as source of inspiration for our work.

A functor \( F : \text{EM}(\mathcal{M}) \to \text{EM}(\mathcal{M}) \) is said to be a lifting of a functor \( F : \mathcal{C} \to \mathcal{C} \) if and only if \( \mathcal{U} \circ F = F \circ \mathcal{U} \). Here, \( \mathcal{U} \) is the forgetful functor \( \mathcal{U} : \text{EM}(\mathcal{M}) \to \mathcal{C} \) mapping an algebra to its carrier. It has a left adjoint \( \mathcal{F} \), mapping an object \( X \in \mathcal{C} \) to the (free) algebra \((\mathcal{M}X, \mu_X)\). We have that \( \mathcal{M} = \mathcal{U} \circ \mathcal{F} \).

Whenever \( F : \mathcal{C} \to \mathcal{C} \) has a lifting \( \mathcal{F} : \text{EM}(\mathcal{M}) \to \text{EM}(\mathcal{M}) \), one has the following functors between categories of coalgebras.

\[
\begin{array}{ccc}
\text{Coalg}_{\mathcal{C}}(\mathcal{F}) & \xrightarrow{\pi} & \text{Coalg}_{\text{EM}(\mathcal{M})}(\mathcal{F}) \\
\end{array}
\]

The functor \( \mathcal{F} \) transforms every coalgebra \( c : S \to F_{\text{MS}}S \) over the base category into a coalgebra \( c^\sharp : FS \to \mathcal{F}FS \). Note that this is a coalgebra on \( \text{EM}(\mathcal{M}) \): the state space carries an algebra, actually the freely generated one, and \( c^\sharp \) is a homomorphism of \( \mathcal{M} \)-algebras.

Intuitively, this amounts to compositionality: like in GSOS specifications, the transitions of a compound state are determined by the transitions of its components.

The functor \( \overline{\mathcal{U}} \) simply forgets about the algebraic structure: \( c^\sharp \) is mapped into

\[
\overline{\mathcal{U}}c^\sharp : \mathcal{MS} = UF_{\text{FS}} \to \overline{\mathcal{U}}F_{\text{FS}} = F_{\text{MS}}.
\]

An important property of \( \overline{\mathcal{U}} \) is that it preserves and reflects behavioural equivalence. On the one hand, this fact usually allows to give concrete characterisation of \( \approx \) for \( \mathcal{F} \)-coalgebras. On the other, it allows, by means of the so-called up-to techniques, to exploit the \( \mathcal{M} \)-algebraic structure of \( \mathcal{F}S \) to check \( \approx \) on \( \overline{\mathcal{U}}c^\sharp \).

By taking \( F = 2 \times (-)^L \) and \( \mathcal{M} = \mathcal{P} \), one transforms \( c : S \to 2 \times (\mathcal{P}S)^L \) into \( \overline{\mathcal{U}}c^\sharp : \mathcal{P}S \to 2 \times (\mathcal{P}S)^L \). The former is a non-deterministic automaton (every \( c \) of this type is a pairing \( \langle o, t \rangle \) of \( o : S \to 2 \), defining the final states, and \( t : S \to \mathcal{P}(S)^L \), defining the transition relation) and the latter is a deterministic automaton which has \( \mathcal{P}S \) as states space. In [53], see also [32], it is shown that, for a certain choice of the lifting \( \mathcal{F} \), this amounts exactly to the standard determinisation from automata theory. This explains why this construction is called the generalised determinisation.

In a sense, this is similar to the translation of probabilistic automata into belief-state transformers that we have seen in Section 2. Indeed, probabilistic automata are coalgebras \( c : S \to (\mathcal{PDS})^L \) and belief state transformers are coalgebras of type \( \mathcal{D}S \to (\mathcal{PDS})^L \). One would like to take \( F = \mathcal{P}^L \) and \( \mathcal{M} = \mathcal{D} \) and reuse the above construction but, unfortunately, \( \mathcal{P}^L \) does not have a suitable lifting to \( \text{EM}(\mathcal{D}) \). This is a consequence of two well known facts: the lack of a suitable distributive law \( \rho : \mathcal{D}P \Rightarrow \mathcal{P}D \) [64] \(^2\) and the one-to-one correspondence between distributive laws and liftings, see e.g. [32]. In the next section, we will nevertheless provide a “powerset-like” functor on \( \text{EM}(\mathcal{D}) \) that we will exploit then in Section 6 to properly model \( \text{PA} \) as belief-state transformers.

\(^2\) As shown in [64], there is no distributive law of the powerset monad over the distribution monad. Note that a “trivial” lifting and a corresponding distributive law of the powerset functor over the distribution monad exists, it is based on [11] and has been exploited in [32]. However, the corresponding “determinisation” is trivial, in the sense that its distribution bisimilarity coincides with bisimilarity, and it does not correspond to the belief-state transformer.
5 Coalgebras on Convex Algebras

In this section we provide several functors on $\text{EM}(\mathcal{D})$ that will be used in the modelling of probabilistic automata as coalgebras over $\text{EM}(\mathcal{D})$. This will make explicit the implicit algebraic structure (convexity) in probabilistic automata and lead to distribution bisimilarity as natural semantics for probabilistic automata in Section 6.

5.1 Convex Powerset on Convex Algebras

We now define a functor, the (nonempty) convex powerset functor, on $\text{EM}(\mathcal{D})$. Let $\mathbb{A}$ be a convex algebra. We define $\mathbb{A}_c$ to be $\mathbb{A}_c = (A_c, a_c)$ where $A_c = \{C \subseteq A \mid C \neq \emptyset, C \text{ is convex}\}$ and $a_c$ is the convex algebra structure given by the following pointwise binary convex combinations: $pC + \bar{p}D = \{pc + \bar{pd} \mid c \in C, d \in D\}$. It is important that we only allow nonempty convex subsets in the carrier $A_c$ of $\mathbb{A}_c$, as otherwise the projection axiom fails.

For convex subsets of a finite dimensional vector space, the pointwise operations are known as the Minkowski addition and are a basic construction in convex geometry, see e.g. [48]. The pointwise way of defining algebras over subsets (carriers of subalgebras) has also been studied in universal algebra, see e.g. [8, 7, 9].

Next, we define $\mathbb{A}_c$ on arrows. For a convex homomorphism $h: \mathbb{A} \rightarrow \mathbb{B}$, $\mathbb{A}_c = \mathbb{B}_c$. The following property ensures that we are on the right track.

$\blacktriangleright$ Proposition 14. $\mathbb{A}_c$ is a convex algebra. If $h: \mathbb{A} \rightarrow \mathbb{B}$ is a convex homomorphism, then so is $\mathbb{A}_c h: \mathbb{A}_c \mathbb{A} \rightarrow \mathbb{A}_c \mathbb{B}$. $\mathbb{A}_c$ is a functor on $\text{EM}(\mathcal{D})$. $\blacktriangleleft$

$\blacktriangleright$ Remark 15. $\mathbb{A}_c$ is not a lifting of $\mathbb{C}$ to $\text{EM}(\mathcal{D})$, but it holds that $\mathbb{C} = \mathbb{U} \circ \mathbb{A}_c \circ \mathbb{F}$ as illustrated below on the left. $\mathbb{A}_c$ is also not a lifting of $\mathbb{P}_{ne}$, the nonempty powerset functor, but we have an embedding natural transformation $\varepsilon: \mathbb{U} \circ \mathbb{A}_c \Rightarrow \mathbb{P}_{ne} \circ \mathbb{U}$ given by $\varepsilon(C) = C$, i.e., we are in the situation:

The right diagram in Remark 15 simply states that every convex subset is a subset, but this fact and the natural transformation $\varepsilon$ are useful in the sequel. In particular, using $\varepsilon$ we can show the next result.

$\blacktriangleright$ Proposition 16. $\mathbb{A}_c$ is a monad on $\text{EM}(\mathcal{D})$, with $\eta$ and $\mu$ as for the powerset monad. $\blacktriangleleft$

5.2 Termination on Convex Algebras

The functor $\mathbb{A}_c$ defined in the previous section allows only for nonempty convex subsets. We still miss a way to express termination. The question of termination amounts to the question of extending a convex algebra $\mathbb{A}$ with a single element $\ast$. This question turns out to be rather involved, beyond the scope of this paper. The answer from [56] is: there are many ways to extend any convex algebra $\mathbb{A}$ with a single element, but there is only one natural functorial way. Somehow now mathematics is forcing us the choice of a specific computational behaviour for termination!
Given a convex algebra $\mathbb{A}$, let $\mathbb{A} + 1$ have the carrier $\mathbb{A} + \{\ast\}$ for $\ast \not\in \mathbb{A}$ and convex operations given by

$$px \oplus \bar{py} = \begin{cases} px + \bar{py}, & x, y \in \mathbb{A}, \ x = \ast \text{ or } y = \ast, \\
\ast, & x = \ast \text{ or } y = \ast. \end{cases} \quad (2)$$

Here, the newly added $\ast$ behaves as a black hole that attracts every other element of the algebra in a convex combination. It is worth remarking that this extension is folklore [23].

**Proposition 17** ([56, 23]). $\mathbb{A} + 1$ as defined above is a convex algebra that extends $\mathbb{A}$ by a single element. The map $h + 1$ obtained with the termination functor in Sets is a convex homomorphism if $h: \mathbb{A} \to \mathbb{B}$ is. The assignments $(-) + 1$ give a functor on EM($\mathcal{D}$).

We call the functor $(-) + 1$ on EM($\mathcal{D}$) the termination functor, due to the following.

**Lemma 18.** The functor $(-) + 1$ is a lifting of the termination functor to EM($\mathcal{D}$).

**Remark 19.** Note that we are abusing notation here: Our termination functor $(-) + 1$ on EM($\mathcal{D}$) is not the coproduct $(-) + 1$ in EM($\mathcal{D}$). The coproduct was concretely described in [33, Lemma 4], and the coproduct $X + 1$ has a much larger carrier than $X + 1$. Nevertheless, we use the same notation as it is very intuitive and due to Lemma 18.

### 5.3 Constant Exponent on Convex Algebras

We now show the existence of a constant exponent functor on EM($\mathcal{D}$). Let $L$ be a set of labels or actions. Let $\mathbb{A}$ be a convex algebra. Consider $\mathbb{A}^L$ with carrier $\mathbb{A}^L = \{f | f: L \to \mathbb{A}\}$ and operations defined (pointwise) by $(pf + \bar{pg})(l) = pf(l) + \bar{pg}(l)$.

The following property follows directly from the definitions.

**Proposition 20.** $\mathbb{A}^L$ is a convex algebra. If $h: \mathbb{A} \to \mathbb{B}$ is a convex homomorphism, then so is $h^L: \mathbb{A}^L \to \mathbb{B}^L$ defined as in Sets. Hence, $(-)^L$ defined above is a functor on EM($\mathcal{D}$).

We call $(-)^L$ the constant exponent functor on EM($\mathcal{D}$). The name and the notation is justified by the following (obvious) property.

**Lemma 21.** The constant exponent $(-)^L$ on EM($\mathcal{D}$) is a lifting of the constant exponent functor $(-)^L$ on Sets.

**Example 22.** Consider a free algebra $\mathcal{F}S = (\mathcal{DS}, \mu)$ of distributions over the set $S$. By applying first the functor $\mathcal{P}_c$, then $(-) + 1$ and then $(-)^L$, one obtains the algebra

$$(\mathcal{P}_c\mathcal{F}S + 1)^L = \begin{pmatrix} \mathcal{D} (CS + 1)^L \\
\alpha \\
(CS + 1)^L \end{pmatrix}$$

where $CS$ is the set of non-empty convex subsets of distributions over $S$, and $\alpha$ corresponds to the convex operations$^3$ $\sum p_i f_i$ defined by

$$(\sum p_i f_i)(l) = \begin{cases} \sum p_i \xi_i, & \xi_i \in f_i(l) \text{ for all } i \in \{1, \ldots, n\} \\
\ast, & f_i(l) = \ast \text{ for some } i \in \{1, \ldots, n\} \end{cases}$$

---

$^3$ In this case, for future reference, it is convenient to spell out the n-ary convex operations.
5.4 Transition Systems on Convex Algebras

We now compose the three functors introduced above to properly model transition systems as coalgebras on \( \text{EM}(D) \). The functor that we are interested in is \( (P_c + 1)^L : \text{EM}(D) \to \text{EM}(D) \). A coalgebra \((S, c)\) for this functor can be thought of as a transition system with labels in \( L \) where the state space carries a convex algebra and the transition function \( c : S \to (P_cS + 1)^L \) is a homomorphism of convex algebras. This property entails compositionality: the transitions of a composite state \( px_1 + px_2 \) are fully determined by the transitions of its components \( x_1 \) and \( x_2 \), as shown in the next proposition. We write \( x \xrightarrow{a} y \) for \( x, y \in S \), the carrier of \( S \) if \( y \in c(x)(a) \), and \( x \not\xrightarrow{a} \) if \( c(x)(a) = \ast \).

\[\begin{align*}
\text{Proposition 23.} & \text{ Let } (S, c) \text{ be a } (P_c + 1)^L\text{-coalgebra, and let } x_1, x_2, y_1, y_2, z \text{ be elements of } S, \text{ the carrier of } S. \text{ Then, for all } p \in (0, 1), \text{ and } a \in L \\
& \text{If } px_1 + px_2 \xrightarrow{\alpha} z \text{ iff } z = py_1 + py_2, x_1 \xrightarrow{\beta} y_1 \text{ and } x_2 \xrightarrow{\gamma} y_2; \\
& px_1 + px_2 \not\xrightarrow{\alpha} \text{ iff } x_1 \not\xrightarrow{\beta} y_1 \text{ or } x_2 \not\xrightarrow{\gamma} y_2.
\end{align*}\]

Transition systems on convex algebras are the bridge between PA and LTSs. In the next section we will show that one can transform an arbitrary PA into a \((P_c + 1)^L\)-coalgebra and that, in the latter, behavioural equivalence coincides with the standard notion of bisimilarity for LTSs (Proposition 27).

6 From PA to Belief-State Transformers

Before turning our attention to PA, it is worth to make a further step of abstraction.

Recall from Remark 15 how \( P_c \) is related to \( C \) and \( P_{ne} \). The following definition is the obvious generalisation.

\[\begin{align*}
\text{Definition 24.} & \text{ Let } M : \text{Sets} \to \text{Sets} \text{ be a monad and } \mathcal{L}_1, \mathcal{L}_2 : \text{Sets} \to \text{Sets} \text{ be two functors. A functor } H : \text{EM}(M) \to \text{EM}(M) \text{ is} \\
& \text{a quasi lifting of } \mathcal{L}_1 \text{ if the diagram on the left commutes.} \\
& \text{a lax lifting of } \mathcal{L}_2 \text{ if there exists an injective natural transformation } e : U \circ H \Rightarrow \mathcal{L}_2 \circ U \\
& \text{as depicted on the right.} \\
& \text{an } (\mathcal{L}_1, \mathcal{L}_2) \text{ quasi-lax lifting if it is both a quasi lifting of } \mathcal{L}_1 \text{ and a lax lifting of } \mathcal{L}_2.
\end{align*}\]

So, for instance, \( P_c \) is a \((C, P_{ne})\) quasi-lax lifting. From this fact, it follows that \((P_c + 1)^L\) is a \(((C + 1)^L, (P_{ne} + 1)^L)\) quasi-lax lifting. Another interesting example is the generalised determinisation (Section 4.2): it is easy to see that \( \overline{F} \) is a \((F, M, F)\)-quasi-lax lifting. Indeed, like in the generalised powerset construction, one can construct the following functors.

\[\begin{align*}
\text{Coalg}_{\text{Sets}}(\mathcal{L}_1) & \xrightarrow{\overline{F}} \text{Coalg}_{\text{EM}(M)}(H) \\
\text{Coalg}_{\text{Sets}}(\mathcal{L}_2) & \xrightarrow{\Pi} \text{Coalg}_{\text{Sets}}(\mathcal{L}_2)
\end{align*}\]

We first define \( \overline{F} \). Take an \( \mathcal{L}_1\)-coalgebra \((S, c)\) and recall that \( \mathcal{F}S \) is the free algebra \( \mu : \mathcal{MMS} \to \mathcal{MS} \). The left diagram in Definition 24 entails that \( HFS \) is an algebra \( \alpha : \mathcal{M}_\mathcal{L}_1S \to \mathcal{L}_1S \). We call \( UC^\sharp \) the composition \( UFS = \mathcal{MS} \xrightarrow{\mathcal{M}_\mathcal{L}_1S} \mathcal{L}_1S = UHFS \). The next lemma shows that \( C^\sharp : \mathcal{FS} \to HFS \) is a map in \( \text{EM}(M) \).
The Power of Convex Algebras

Lemma 25. There is a 1-1 correspondence between $\mathcal{L}_1$-coalgebras on $\text{Sets}$ and $\mathcal{H}$-coalgebras on $\text{EM}(\mathcal{M})$ with carriers free algebras:

$$c: S \to \mathcal{L}_1 S \in \text{Sets}$$

$$c^\#: F S \to \mathcal{H} F S \in \text{EM}(\mathcal{M})$$

given $c$, we have $U c^\# = \alpha \circ M c$ for $\alpha = \mathcal{H} F S$.

Proposition 26. The assignment $\mathcal{T}(S, c) = (F S, c^\#)$ and $\mathcal{T}(h) = M h$ gives a functor $\mathcal{T}: \text{Coalg}_{\text{Sets}}(\mathcal{L}_1) \to \text{Coalg}_{\text{EM}(\mathcal{M})}(\mathcal{H})$.

Now we can define $\mathcal{U}: \text{Coalg}_{\text{EM}(\mathcal{M})}(\mathcal{H}) \to \text{Coalg}_{\text{Sets}}(\mathcal{L}_2)$ as mapping every coalgebra $(S, c)$ with $c: S \to \mathcal{H} S$ into

$$\mathcal{U}(S, c) = (U S, e_S \circ U c)$$

and every coalgebra homomorphism $h: (S, c) \to (T, d)$ into $\mathcal{U} h = U h$. Routine computations confirm that $\mathcal{U}$ is a functor.

Since $\mathcal{U}$ is a functor that keeps the state set constant and is identity on morphisms, every kernel bisimulation on $(S, c)$ is also a kernel bisimulation on $\mathcal{U}(S, c)$. The converse is not true in general: a kernel bisimulation $R$ on $\mathcal{U}(S, c)$ is a kernel bisimulation on $(S, c)$ only if it is a congruence with respect to the algebraic structure of $S$.

Formally, $R$ is a congruence if and only if the set $U S / R$ of equivalence classes of $R$ carries an Eilenberg-Moore algebra and the function $U [-]_R: U S \to U S / R$ mapping every element of $U S$ to its $R$-equivalence class is an algebra homomorphism.

Proposition 26. The following are equivalent:

1. $R$ is a kernel bisimulation on $(S, c)$,
2. $R$ is a congruence of $(S, c)$ and a kernel bisimulation of $\mathcal{U}(S, c)$.

In particular, Proposition 26 and the following result ensure that the functor $\mathcal{U}: \text{Coalg}_{\mathcal{D}(\mathcal{D}^L)}(\mathcal{P}_c + 1)^L \to \text{Coalg}_{\text{Sets}}\mathcal{P}^L$ preserves and reflects $\approx$.

Proposition 27. Let $(S, c)$ be a $(\mathcal{P}_c + 1)^L$-coalgebra. Behavioural equivalence on $\mathcal{U}(S, c)$ is a convex congruence\(^4\). Hence, $\mathcal{U}$ preserves and reflects behavioural equivalence.

This means that $\approx$ for $(\mathcal{P}_c + 1)^L$-coalgebras, called transition systems on convex algebras in Section 5.4, coincides with the standard notion of bisimilarity for LTSs.

Table 1 summarises all models of PA: from the classical model $M$ being a $\mathcal{P}D^L$-coalgebra $(S, c_M)$ on $\text{Sets}$, via the convex model $M_c$ obtained as $\mathcal{T}_{\text{conv}}(S, c_M)$, to the belief state transformer $M_b$. The latter coincides with $\mathcal{U} \circ \mathcal{T} \circ \mathcal{T}_{\text{conv}}(S, c_M)$.

Theorem 28. Let $(S, c_M)$ be a probabilistic automaton. For all $\xi, \zeta \in \mathcal{D} S$,

$$\xi \sim_d \zeta \quad \Leftrightarrow \quad \xi \approx \zeta \quad \text{in} \quad \mathcal{U} \circ \mathcal{T} \circ \mathcal{T}_{\text{conv}}(S, c_M).$$

Hence, distribution bisimilarity is indeed behavioural equivalence on the belief-state transformer and it coincides with standard bisimilarity.

\(^4\) Convex congruences are congruences of convex algebras, see e.g. [55]. They are convex equivalences, i.e., closed under componentwise-defined convex combinations.
Table 1 The three PA models, their corresponding Sets-coalgebras, and relations to $M$.

<table>
<thead>
<tr>
<th>$M$ = $(S, L, \to)$</th>
<th>$M_e$ = $(S, L, \to e)$</th>
<th>$M_{bs}$ = $(DS, L, \to)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(S, cM) : S \to (PD)^L$</td>
<td>$(S, \hat{c}M) : S \to (C + 1)^L$</td>
<td>$(DS, \hat{c}M) : DS \to (PD(S))$</td>
</tr>
<tr>
<td>$c_M$</td>
<td>$\hat{c}_M = conv^L \circ c_M$</td>
<td>$\hat{c}<em>M = (e</em>{FS} + 1)^L \circ U_M^2$</td>
</tr>
</tbody>
</table>

7 Bisimulations Up-To Convex Hull

As we mentioned in Section 4.2, the generalised determinisation allows for the use of up-to techniques [42, 45]. An important example is shown in [6]: given a non-deterministic automaton $c : S \to 2 \times P(S)^L$, one can reason on its determinisation $U^2 : P(S) \to 2 \times P(S)^L$ up-to the algebraic structure carried by the state space $P(S)$. Given a probabilistic automaton $(S, L, \to)$, we would like to exploit the algebraic structure carried by $D(S)$ to prove properties of the corresponding belief states transformer $(D(S), L, \to)$. Unfortunately, the lack of a suitable distributive law [64] makes it impossible to reuse the abstract results in [5]. Fortunately, we can redo all the proofs by adapting the theory in [45] to the case of probabilistic automata.

Hereafter we fix a PA $M = (S, L, \to)$ and the corresponding belief states transformer $M_{bs} = (D(S), L, \to)$. We denote by $Rel_{D(S)}$ the lattice of relations over $D(S)$ and define the monotone function $b : Rel_{D(S)} \to Rel_{P(S)}$ mapping every relation $R \in Rel_{D(S)}$ into

$$b(R) := \{ (\zeta_1, \zeta_2) \mid \forall a \in L, \forall \zeta_1' s.t. \, \zeta_1 \looparrowright a \zeta_1', \exists \zeta_2' s.t. \, \zeta_2 \looparrowright a \zeta_2', \exists \zeta_1'' s.t. \, \zeta_1 \looparrowright a \zeta_1'' \text{ and } (\zeta_1', \zeta_2') \in R, \forall \zeta_1'' s.t. \, \zeta_2 \looparrowright a \zeta_1'' \text{ and } (\zeta_1', \zeta_1'') \in R \}.$$ 

A bisimulation is a relation $R$ such that $R \subseteq b(R)$. Observe that these are just regular bisimulations for labeled transition systems and that the greatest fixpoint of $b$ coincides exactly with $\sim_d$. The coinduction principle informs us that to prove that $\zeta_1 \sim_d \zeta_2$ it is enough to exhibit a bisimulation $R$ such that $(\zeta_1, \zeta_2) \in R$.

Example 29. Consider the PA in Figure 1 (left) and the belief-state transformer generated by it (right). It is easy to see that the (Dirac distributions of the) states $x_2$ and $y_2$ are in $\sim_{d'}$: the relation $\{(x_2, y_2)\}$ is a bisimulation. Also $\{(x_3, y_3)\}$ is a bisimulation: both $x_3 \xrightarrow{p^\#} y_3$ and $y_3 \xrightarrow{p^\#} x_3$. More generally, for all $\zeta, \xi \in D(S), p, q \in [0, 1]$, $p\zeta + p\xi \sim_d q\zeta + q\xi$ since both $p\zeta + p\xi \sim_d \hat{q}_3 \xrightarrow{\hat{y}}$ and $q\zeta + q\xi \sim_d \hat{y}$.

$$p\zeta + p\xi \sim_d q\zeta + q\xi \text{ since both } p\zeta + p\xi \sim_d \hat{q}_3 \xrightarrow{\hat{y}} \text{ and } q\zeta + q\xi \sim_d \hat{y}. \tag{3}$$

Proving that $x_0 \sim_d y_0$ is more complicated. We will show this in Example 32 but, for the time being, observe that one would need an infinite bisimulation containing the following pairs of states.

$$x_0 \xrightarrow{a} x_1 \xrightarrow{a} \frac{1}{2} x_1 + \frac{1}{2} x_2 \xrightarrow{a} \frac{1}{3} x_1 + \frac{1}{3} x_1 + \frac{1}{3} x_2 \xrightarrow{a} \ldots$$

$$y_0 \xrightarrow{a} \frac{1}{2} y_1 + \frac{1}{2} y_2 \xrightarrow{a} \frac{1}{4} y_1 + \frac{3}{4} y_2 \xrightarrow{a} \frac{1}{5} y_1 + \frac{4}{5} y_2 \xrightarrow{a} \ldots$$

Indeed, all the distributions depicted above have infinitely many possible choices for $\xrightarrow{a}$ but, whenever one of them executes a depicted transition, the corresponding distribution is forced, because of (3), to also choose the depicted transition.
An up-to technique is a monotone map \( f: \text{Rel}_{D(S)} \to \text{Rel}_{D(S)} \), while a bisimulation up-to \( f \) is a relation \( R \) such that \( R \subseteq b f(R) \). An up-to technique \( f \) is said to be sound if, for all \( R \in \text{Rel}_{D(S)} \), \( R \subseteq b f(R) \) entails that \( R \subseteq \sim_d \). It is said to be compatible if \( f(b(R)) \subseteq b f(R) \). In [45], it is shown that every compatible up-to technique is also sound.

Hereafter we consider the convex hull technique \( \text{conv} \): \( \text{Rel}_{D(S)} \to \text{Rel}_{D(S)} \) mapping every relation \( R \in \text{Rel}_{D(S)} \) into its convex hull which, for the sake of clarity, is

\[
\text{conv}(R) = \{(p\zeta_1 + p\xi_1, p\zeta_2 + p\xi_2) \mid (\zeta_1, \zeta_2) \in R, (\xi_1, \xi_2) \in R \text{ and } p \in [0, 1]\}.
\]

\[\begin{align*}
&\text{Proposition 30.} \; \text{conv is compatible.} \\
\end{align*}\]

This result has two consequences: First, \( \text{conv} \) is sound\(^5\) and thus one can prove \( \sim_d \) by means of bisimulation up-to \( \text{conv} \); Second, \( \text{conv} \) can be effectively combined with other compatible up-to techniques (for more details see [45] or the full version). In particular, by combining \( \text{conv} \) with up-to equivalence – which is well known to be compatible – one obtains up-to congruence \( \text{cgr}: \text{Rel}_{D(S)} \to \text{Rel}_{D(S)} \). This technique maps a relation \( R \) into its congruence closure: the smallest relation containing \( R \) which is a congruence.

\[\begin{align*}
&\text{Proposition 31.} \; \text{cgr is compatible.} \\
\end{align*}\]

Since \( \text{cgr} \) is compatible and thus sound, we can use bisimulation up-to \( \text{cgr} \) to check \( \sim_d \).

\[\begin{align*}
&\text{Example 32.} \; \text{We can now prove that, in the PA depicted in Figure 1, } x_0 \sim_d y_0. \; \text{It is easy to see that the relation } R = \{(x_2, y_2), (x_3, y_1), (x_1, \frac{1}{2}y_1 + \frac{1}{2}y_2), (x_0, y_0)\} \text{ is a bisimulation up-to } \text{cgr}: \text{consider } (x_1, \frac{1}{2}y_1 + \frac{1}{2}y_2) \text{ (the other pairs are trivial) and observe that} \\
&\begin{align*}
&x_1 \xrightarrow{a} \frac{1}{2}x_1 + \frac{1}{2}x_2 \\
&x_1 \xrightarrow{\text{cgr}(R)} \frac{1}{2}x_3 + \frac{1}{2}x_2 \\
&\frac{1}{2}y_1 + \frac{1}{2}y_2 \xrightarrow{a} \frac{1}{2}y_1 + \frac{3}{2}y_2 \\
&\frac{1}{2}y_1 + \frac{1}{2}y_2 \xrightarrow{\text{cgr}(R)} \frac{1}{2}y_3 + \frac{1}{2}y_2 \\
\end{align*}
\]

Since all the transitions of \( x_1 \) and \( \frac{1}{2}y_1 + \frac{1}{2}y_2 \) are obtained as convex combination of the two above, the arriving states are forced to be in \( \text{cgr}(R) \). In symbols, if \( x_1 \xrightarrow{\alpha} \zeta = p(\frac{1}{2}x_1 + \frac{1}{2}x_2) + p(\frac{1}{2}x_3 + \frac{1}{2}x_2) \), then \( \frac{1}{2}y_1 + \frac{1}{2}y_2 \xrightarrow{\alpha} \xi = p(\frac{1}{2}y_1 + \frac{1}{2}y_2) + p(\frac{1}{2}y_3 + \frac{1}{2}y_2) \) and \( (\zeta, \xi) \in \text{cgr}(R) \).

Recall that in Example 29, we showed that to prove \( x_0 \sim_d y_0 \) without up-to techniques one would need an infinite bisimulation. Instead, the relation \( R \) in Example 32 is a finite bisimulation up-to \( \text{cgr} \). It turns out that one can always check \( \sim_d \) by means of only finite bisimulations up-to. The key to this result is the following theorem, recently proved in [55].

\[\begin{align*}
&\text{Theorem 33.} \; \text{Congruences of finitely generated convex algebras are finitely generated.} \\
\end{align*}\]

This result informs us that for a PA with a finite state space \( S, \sim_d \subseteq D(S) \times D(S) \) is finitely generated (since \( \sim_d \) is a congruence, see Proposition 27). In other words there exists a finite relation \( R \) such that \( \text{cgr}(R) = \sim_d \). Such \( R \) is a finite bisimulation up-to \( \text{cgr} \):

\[
R \subseteq \text{cgr}(R) = \sim_d = b(\sim_d) = b(\text{cgr}(R)).
\]

\[\begin{align*}
&\text{Corollary 34.} \; \text{Let } (S, L, \to) \text{ be a finite PA and let } \zeta_1, \zeta_2 \in D(S) \text{ be two distributions such that } \zeta_1 \sim_d \zeta_2. \; \text{Then, there exists a finite bisimulation up-to } \text{cgr} \text{ relation } R \text{ such that } (\zeta_1, \zeta_2) \in R. \\
\end{align*}\]

\(^5\) In [47] a similar up-to technique called “up-to lifting” is defined in the context of probabilistic \(\lambda\)-calculus and proven sound.
8 Conclusions and Future Work

Belief-state transformers and distribution bisimilarity have a strong coalgebraic foundation which leads to a new proof method – bisimulation up-to convex hull. More interestingly, and somewhat surprisingly, proving distribution bisimilarity can be achieved using only finite bisimulation up-to witness. This opens exciting new avenues: Corollary 34 gives us hope that bisimulations up-to may play an important role in designing algorithms for automatic equivalence checking of PA, similar to the one played for NDA. Exploring their connections with the algorithms in [26, 20] is our next step.

From a more abstract perspective, our work highlights some limitations of the bialgebraic approach [62, 3, 34]. Despite the fact that our structures are coalgebras on algebras, they are not bialgebras: but still $\approx$ is a congruence and it is amenable to up-to techniques. We believe that laz bialgebra may provide some deeper insights.

References


Refinement for Signal Flow Graphs

Filippo Bonchi∗, Joshua Holland, Dusko Pavlović†, and Paweł Sobociński‡

1 Ecole Normale Supérieure, Lyon, France
2 University of Southampton, Southampton, UK
3 University of Hawaii at Manoa, Honolulu, Hawaii, US
4 University of Southampton, Southampton, UK

Abstract

The symmetric monoidal theory of Interacting Hopf Algebras provides a sound and complete axiomatisation for linear relations over a given field. As is the case for ordinary relations, linear relations have a natural order that coincides with inclusion. In this paper, we give a presentation for this ordering by extending the theory of Interacting Hopf Algebras with a single additional inequation. We show that the extended theory gives rise to an abelian bicategory—a concept due to Carboni and Walters—and highlight similarities with the algebra of relations. Most importantly, the ordering leads to a well-behaved notion of refinement for signal flow graphs.

1998 ACM Subject Classification F.3.1 Specifying and Verifying and Reasoning about Programs, F.3.2 Semantics of Programming Languages

Keywords and phrases Signal flow graphs, refinement, operational semantics, string diagrams, symmetric monoidal inequality theory

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2017.24

1 Introduction

Signal Flow Graphs (SFGs) were introduced in the 1940s by Shannon [19] as a formal circuit model of a class of simple analog computing machines. They are a common abstraction in control theory and signal processing, used for modelling physical systems and their controllers. Nowadays, cyber-physical systems are modelled and simulated in graphical environments such as Simulink and Modelica that can be seen as great-grandchildren of SFGs.

Their ubiquity is merited because SFGs serve both as processors of analogue signals (analytic functions) in continuous time, and as stream transducers in discrete time. The latter makes them amenable to techniques developed by computer scientists for programming language semantics. For instance Rutten [18] showed that coinduction, just as in process algebra, provides a useful proof principle for SFGs. Another example is the signal flow calculus [6] where SFGs are represented using string diagrammatic syntax equipped with both a structural operational semantics and a denotational semantics in terms of linear relations. Most importantly, denotational equality, which by full abstraction [6] coincides with observational equivalence (trace equivalence), enjoys a sound and complete axiomatization [4].

The same equational theory was independently proposed by Baez and Erbele [3].

∗ Partially supported by the project ANR-16-CE25-0011 REPAS and AFOSR.
† Partially supported by NSF and AFOSR.
‡ Partially supported by AFOSR.
The axiomatisation of [4, 3]—a symmetric monoidal theory (SMT) whose terms are typically rendered graphically as string diagrams—is the starting point of the present work. We adopt the terminology of [5]: the theory of interacting Hopf algebras over a ring \( R \), denoted \( \text{IH}_R \), consists of a pair of monoids (distinguished by black and white colouring) and a pair of comonoids (again, black and white). These black-white (co)monoids satisfy the equations of Frobenius and Hopf algebras, individually recalled in Examples 3 and 4, as illustrated in the schematic to the right.

A theorem in [4] states that \( \text{IH}_R \) is a presentation for \( \text{LinRel}_k \), the category with arrows linear relations (a.k.a. additive relations) over \( k \), the field of fractions of \( R \): relations that are also linear subspaces. This paved the way for an equational study of elementary linear algebra by means of string diagrams that, in [22], became graphical linear algebra.

Like relations, linear relations are equipped with an ordering that plays a pivotal role in many applications. It is therefore worth seeing \( \text{LinRel}_k \) not as a mere category but rather as a poset enriched category. In this work, we provide a presentation for the underlying posetal structure of \( \text{LinRel}_k \). Our main result states that it is enough to add a single inequation

\[
\circ \leq \bullet
\]

to the equational theory of \( \text{IH}_R \) in order to obtain a sound and complete axiomatisation of the ordering between the arrows of \( \text{LinRel}_k \). Viewed as linear relations, (2) says that the unique zero-dimensional subspace \( \{0\} \) of \( k \), considered as a vector space over itself, is a subset of the unique one-dimensional subspace. Of course, the reverse inequality does not hold.

The focus on the order sheds lights on some interesting properties of \( \text{IH}_R \). We show that \( \text{IH}_R \) forms an abelian bicategory [11, Def. 5.1] and that it supports operations akin to the algebra of relations [14]. Moreover, the order resolves a mystery surrounding the equational theory of interacting Hopf algebras. The system summarised in (1) is symmetric. There is no difference, equationally, between the white (co)monoid and the black (co)monoid, in spite their different meaning as linear relations: the white is the additive structure, while the black is “copying”, e.g. \( \bullet \) is typically the diagonal relation. Crucially, (2) breaks this symmetry.

When \( R = k[x] \) (the ring of polynomials with indeterminate \( x \) and coefficients from field \( k \)), \( \text{IH}_R \) provides a sound and complete axiomatisation for SFGs [5]. The addition of (2) gives a sound and complete axiomatisation for what we call refinement of SFGs.

1.1 Structure of the paper

The problem of refinement of SFGs is informally explained with an example in Section 2. In Section 3 we recall the basic concepts of SMTs and, in Section 4, the theory of Interacting Hopf Algebras. In Section 5 we extend the concept of monoidal theory to handle inequations. Section 6 is devoted to proving our main result and, in Section 7, we shed light on the algebraic structure of the resulting theory, drawing parallels with relational algebra. Finally, in Section 8 we return to our motivating problem and discuss related work in Section 9.

2 Fibonacci’s rabbits and guinea pigs

A signal flow graph of sort \((m, n)\), using the discrete semantics, is a stream transducer that takes \( m \) input streams and produces \( n \) output streams. For example, consider the \((1,1)\) SFG
below, which implements the well-known Fibonacci recurrence relation:

\[
\text{fFib} := \begin{array}{c}
\text{\textbullet} \\
\text{\textbullet} \\
\text{\textbullet} \\
\text{\textbullet} \\
\text{\textbullet} \\
\end{array}
\]

The white circles are adders (two inputs and one output), the black circles are duplicators (one input and two outputs). The ‘\(x\)’ gates are delays, or one-state buffers, which we assume to be initialised with zero. Given the sequence of inputs 1; 0; 0; 0; 0; 0; ... , the output is the Fibonacci sequence 1; 2; 3; 5; 8; 13; ... . We illustrate the first few steps below: the state of each delay is illustrated by the number above it, the remaining numbers keep track of the value on each wire at each iteration. A formal operational semantics is recalled in Section 8.

\[
\text{rFib} := \begin{array}{c}
\text{\textbullet} \\
\text{\textbullet} \\
\text{\textbullet} \\
\text{\textbullet} \\
\text{\textbullet} \\
\end{array}
\]

This output—according to the Fibonacci’s rule [20] for rabbit reproduction—is the total number of rabbit pairs in each month, starting with a pair of rabbits (the first input is 1), and subsequently not adding nor taking away pairs (all further inputs are 0). Other inputs are possible, in this sense generalising Fibonacci; e.g. adding a pair for two months and taking away two every third month (input 1; 1; 2; 1; 1; 2; ... ) yields (1; 3; 3; 5; 10; 14; ... ).

A trace of an \((m, n)\) SFG \(c\) is a pair \((\alpha, \beta)\) where \(\alpha\) is an \(m\)-tuple and \(\beta\) is the output \(n\)-tuple produced by \(c\) on \(\alpha\): e.g. (3) has (1; 0; 0; ... , 1; 2; 3; ... ) and (1; 1; −2; ... , 1; 3; 3; ... ) as traces. The behaviour of a signal flow graph is the set of all its traces. Note that behaviour is a functional relation on streams; in particular, if an SFG is invertible then its inverse has the opposite relation as behaviour. Here (3) is invertible and has the following inverse, where the ‘−1 gates’, instances of amplifiers, multiply their input by −1:

\[
\text{rFib}^{-1} := \begin{array}{c}
\text{\textbullet} \\
\text{\textbullet} \\
\text{\textbullet} \\
\text{\textbullet} \\
\text{\textbullet} \\
\end{array}
\]

The SFG above thus solves the toy sustainable rabbit farming problem: how many rabbits must the farmer buy and sell in each month to maintain, say, four pairs in her rabbit pen? The answer is obtained by using 4; 4; 4; 4; 4; ... as input to (5), resulting in 4; −4; 0; −4; 0; ... : i.e. four pairs bought in the first month and, subsequently, four pairs sold every 2nd month.

The proof that \(\text{rFib}^{-1}\) is the inverse of \(\text{rFib}\) consists of an algebraic manipulation of string diagrams: we shall demonstrate this below, after a brief discussion of the mathematics behind the approach. As explained in the Introduction, the theory \(\mathbb{IH}_R\) of Interacting Hopf Algebras characterises linear relations, and an example of such a relation is the behaviour of any signal flow graph when \(R = k[x]\). This algebraic theory is not a classical (finite product) algebraic theory but a symmetric monoidal theory. This means replacing traditional tree-like syntax with string diagrams. Concretely, \(\mathbb{IH}_k[x]\) involves two commutative monoids and two commutative comonoids, meaning that string diagrams are built up from the following:

\[
\text{\textbullet} \\
\text{\textbullet} \\
\text{\textbullet} \\
\text{\textbullet} \\
\text{\textbullet} \\
\end{array}
\]

where \(k\) ranges over the coefficients in \(k\). The different colouring given to the indeterminate \(x\) is inspired by its special semantic role in SFGs.
The Fibonacci SFG (3), and its inverse (5)—as string diagrams—look as follows:

\[ \text{fFib} := \begin{array}{c}
\text{X} \quad \text{X} \\
\text{X} \quad \text{X}
\end{array} \]
\[ \text{rFib} := \begin{array}{c}
\text{X} \quad \text{X} \\
\text{X} \quad \text{X}
\end{array} \]

Note the similarity with SFGs (3) and (5) – the string diagram, roughly speaking, is obtained by forgetting the direction of flow—i.e. erasing the arrowheads—and by replacing U-turn wires with so-called ‘cups’ and ‘caps’, formed by composing multiplication with counit, and unit with comultiplication. We use notation \( \text{fFib}, \text{rFib} \) to emphasise that (undirected) string diagrams have been obtained by translating SFGs that have a left-to-right signal flow.

Using the equational theory of [4, 3] we show—by diagrammatic reasoning, i.e. algebraic manipulation of string diagrams—that the inverse of the Fibonacci SFG is as claimed:

\[ \text{fFib} = \begin{array}{c}
\text{X} \quad \text{X} \\
\text{X} \quad \text{X}
\end{array} \]
\[ \text{rFib} = \begin{array}{c}
\text{X} \quad \text{X} \\
\text{X} \quad \text{X}
\end{array} \]

\[ \text{rFib}(7) \]

Note that \( \text{fFib} \) is equal to the mirror image of its inverse \( \text{rFib} \): this, operationally speaking, means simply that the behaviours of \( \text{fFib} \) and \( \text{rFib} \) are opposites as relations. This worked example demonstrates the power of equational reasoning in \( \mathbb{I}_{\mathbb{H}_k[x]} \).

Let us now consider a more interesting variant of the sustainable farming problem. Suppose that the rabbit farmer also keeps guinea pigs, which, for the sake of this paper, gestate twice as long as rabbits. The SFG for guinea pigs is the following:

\[ \text{fGui} := \begin{array}{c}
\text{X} \quad \text{X} \\
\text{X} \quad \text{X}
\end{array} \]

For instance, on input \( 1; 0; 0; 0; 0; 0; 0; \ldots \) the output is \( 1; 1; 2; 2; 3; 3; 5; \ldots \). The combined rabbit and guinea pig pen has SFG (8(i)), with two inputs and one output. The inputs mean buying or selling a species; the output is the total number of animals in the pen.

We now pose the sustainable farming problem: how many rabbits and guinea pigs must the farmer buy and sell in order to keep the total population fixed? To solve the problem, we draw the string diagram that serves as the specification of the inverse of (8(i)), as in (8(ii)). Now \( \text{fFib} \) and \( \text{fGui} \) are invertible so we can replace (8(ii)) with the equal diagram (8(iii)).

This is as far as we can go in our quest for a SFG, since unfortunately, the specification is not functional: there is no unique solution to the sustainable farming problem. One simple solution is to divide the pen in half and limit the species separately. This, as a SFG, is (8(iv)). As string diagrams (8(iii)) and (8(iv)) are not equal. Yet every trace of the second is a trace of the first; this is an example of refinement. In this paper, we extend \( \mathbb{I}_{\mathbb{H}} \) and characterise this refinement relation. Indeed, we will see that, as string diagrams, (8(iv)) \( \leq \) (8(iii)).
3 Symmetric Monoidal Theories and props

A symmetric monoidal theory (SMT) \((\Sigma, E)\) consists of a set \(\Sigma\) of generators \(o : m \rightarrow n\), each with an arity \(m\) and coarity \(n\) \((m, n \in \mathbb{N})\), along with a set \(E\) of equations, which are pairs \((t_1, t_2 : m \rightarrow n)\) of \(\Sigma\)-terms; \(t_1\) and \(t_2\) must have the same arity and coarity. A \(\Sigma\)-term is constructed inductively from generators in \(\Sigma\), together with the identity \(\text{id} : 1 \rightarrow 1\) and the symmetry \(\sigma_{1,1} : 2 \rightarrow 2\), using composition \(\circ\) and monoidal product \(\oplus\). Given \(\Sigma\)-terms \(t : k \rightarrow l\), \(u : l \rightarrow m\) and \(v : m \rightarrow n\), we construct \(\Sigma\)-terms \(t \circ u : k \rightarrow m\) and \(t \oplus v : k \oplus n \rightarrow l \oplus n\).

\(\Sigma\)-terms are rendered as diagrams and are considered up to the laws of symmetric strict monoidal categories. Analogously to SFGs, generators are drawn as “circuit components” with dangling wires. The identity is drawn — and the symmetry \(\sigma\). Composition of terms is placing them side-by-side and joining the wires. The monoidal product \(\oplus\) is stacking terms on top of each other, as in the following examples. Next we introduce some important SMTs, which are used as building blocks to construct the full SMT for reasoning about SFGs.

▶ Example 1 (The SMT \((\Sigma_M, E_M)\) of commutative monoids). \(\Sigma_M\) contains two generators: multiplication \(\circ : 2 \rightarrow 1\) and unit \(\circ : 0 \rightarrow 1\). \(E_M\) contains three equations; we show them both with composition and product explicitly and as diagrams.

\[
\begin{align*}
(\circ \circ \text{id}) : & \rightarrow = \text{id} \\
\circ & = \sigma_{1,1} : \rightarrow \\
(\circ \circ \text{id}) : & = (\text{id} \circ) ; \\
\circ & = \circ
\end{align*}
\]

(9)

▶ Example 2 (The SMT \((\Sigma_C, E_C)\) of commutative comonoids). Again there are two generators, but this time mirrored: there is a comultiplication \(\imath : 1 \rightarrow 2\) and a counit \(\imath : 1 \rightarrow 0\). The equations are the following, this time given only in the diagrammatic form:

\[
\begin{align*}
\circ & = \circ \\
\circ & = \circ \\
\circ & = \circ
\end{align*}
\]

(10)

Hopf and Frobenius (bi)monoids are two important ways that monoids and comonoids interact; both are needed for the theories we define in this paper.

▶ Example 3 (R Hopf monoids). The generators of the SMT are simply those in \(\Sigma_M \cup \Sigma_C\), and to the equations in \(E_M\) and \(E_C\) we add the following bimonoid laws:

\[
\begin{align*}
\circ & = \circ \\
\circ & = \circ \\
\circ & = \circ \\
\circ & = \text{id}_0
\end{align*}
\]

(11)

For a commutative ring \(R\), we need to add generators \(-k\) for every \(k \in R\) and stipulate

\[
\begin{align*}
(1) & = - \\
(k_1)(k_2) & = (k_1k_2) \\
(0) & = \circ \\
\frac{k_1}{k_2} & = \frac{k_1}{k_2}
\end{align*}
\]

(12)

▶ Example 4 ((extra special) Frobenius monoids). The generators are \(\Sigma_M \cup \Sigma_C\). Keeping (1) in mind, we colour all the generators in grey, which will later be instantiated as either black or white. Our equations are now the Frobenius law, together with the special and extra equations. We often call the latter the “bone” equation, due to its appearance when drawn.

\[
\begin{align*}
- & = \circ \\
\circ & = \circ \\
\text{id}_0 & = \circ
\end{align*}
\]

(13)
We can obtain a symmetric monoidal category from an SMT \((\Sigma, E)\) as follows:

- objects are natural numbers
- arrows \(m \to n\) are \(\Sigma\)-terms \(m \to n\) modulo the laws of symmetric monoidal categories
- and the (smallest congruence containing) the equations \(t_1 = t_2\) for each pair \((t_1, t_2) \in E\)

Such a category is a special type of symmetric monoidal category called a prop.

**Definition 5.** A prop (product and permutation category) is a strict symmetric monoidal category with objects \(\mathbb{N}\), where \(m \oplus n := m + n\). A homomorphism is an identity-on-objects symmetric monoidal functor, giving a category \(PROP\).

**Example 6.** Given a commutative ring \(R\), the prop \(\text{Mat}_R\) of matrices over \(R\) has as arrows \(m \to n\) the \(n \times m\) matrices, composition ; is matrix multiplication and \(A \oplus B\) is the matrix \(\begin{pmatrix} A & 0 \\ 0 & B \end{pmatrix}\). In [16, 7] it is shown that \(\text{Mat}_R\) is isomorphic to the prop \(\mathbb{H}A_R\) arising from the SMT of Hopf monoids over \(R\) (Example 3). The isomorphism \(S' : \mathbb{H}A_R \to \text{Mat}_R\) maps

\[
\begin{align*}
- & \mapsto \begin{pmatrix} 1 \\ 1 \end{pmatrix} \\
- & \mapsto \begin{pmatrix} k \\ 1 \end{pmatrix} \\
- & \mapsto \begin{pmatrix} k \\ 1 \end{pmatrix} \\
- & \mapsto \begin{pmatrix} 1 \\ 1 \end{pmatrix} \\
- & \mapsto ! \quad (14)
\end{align*}
\]

where \(! : 0 \to 1\) and \(j : 1 \to 0\) are given by the universal property of \(0\) in \(\text{Mat}_R\).

Observe (14) defines \(S'\) for all arrows \(A\) of \(\mathbb{H}A_R\). More generally, to specify a homomorphism from a prop obtained from an SMT \((\Sigma, E)\), it is enough to define it on the generators in \(\Sigma\), and check that the equations \(E\) hold in the image. We shall often use this argument.

## 4 Interacting Hopf Algebras

Zanasi with the first and third authors introduced the SMT of Interacting Hopf Algebras [5, 7] as a foundation for SFGs [4, 6, 8]. We recall the equational theory in Fig. 1 where \(R\) is a
fixed principal ideal domain, and denote the resulting prop by $\mathbb{I}R$. As illustrated in (1), the theory features monoids and comonoids that interact either as extra special Frobenius monoids (Example 4) or as Hopf monoids (Example 3). One remarkable feature of this equational theory is that it contains two symmetries: (i) that the mirror image ($\dagger$) of any equation is an equation, and (ii) is that the photographic negative ($\circ$, white $\leftrightarrow$ black) of any equation is an equation. Formally, define prop morphisms $(-)^\dagger: \mathbb{I}R^{\text{op}} \to \mathbb{I}R$ mapping $\mapsto \mapsto k^0 \mapsto \mapsto \mapsto \mapsto \mapsto \mapsto \mapsto 0$ and $(-)^\circ: \mathbb{I}R \to \mathbb{I}R$ mapping $\mapsto \mapsto \mapsto \mapsto k^0 \mapsto \mapsto \mapsto \mapsto \mapsto \mapsto 0$ (15)

The morphism $(-)^\dagger$ is related to the self-dual compact closed structure [15] of $\mathbb{I}R$ defined for each $n \in \mathbb{N}$ by assigning $\eta_n: 0 \to n + n$ (cap) and $\epsilon_n: n + n \to 0$ (cup) the string diagrams:

$$\eta_n = \begin{array}{c}
\text{n} \\
\text{n}
\end{array}, \epsilon_n = \begin{array}{c}
n \\
\text{n}
\end{array}, 0 = \text{id}_0, n + 1 = \begin{array}{c}
n + n
\end{array}$$

**Remark 7.** Above we used $\begin{array}{c}
n \\
\text{n}
\end{array}$ for the $n$-fold monoidal product of $\mapsto$, $\begin{array}{c}
n \\
\text{n}
\end{array}$ for id$_n$ and $\begin{array}{c}
n \\
\text{n}
\end{array}$. The same convention will be used for the white structure. We shall omit the labels when there is no risk of ambiguity.

The contravariant morphism induced by the compact closed structure coincides with $(-)^\dagger$, as defined in (15), that is for all $A : m \to n$:

$$\begin{array}{c}
n \text{m}
\end{array} = \begin{array}{c}
n \text{m}
\end{array}$$

Consider the prop $\text{LinRel}_k$ of linear relations over $k$, the field of fractions of $R$. An arrow $m \to n$ is a linear subspace of $k^m \times k^n$, is relational composition and $\oplus$ is direct sum. Also $\text{LinRel}_k$ has a self-dual compact closed structure: the induced contravariant morphism $(-)^\dagger: \text{LinRel}_k^{\text{op}} \to \text{LinRel}_k$ maps a linear relation $R \subseteq k^m \times k^n$ to its opposite $R^\dagger \subseteq k^n \times k^m$. Let $*$ be the unique element of the 0 dimensional vector space $k^0$. Then

$$\begin{array}{c}
\mapsto \{(x, y) \mid x, y \in k\} & \mapsto \{(\ast, 0)\}
\end{array} \begin{array}{c}
\mapsto \{(x, kx) \mid x \in k\}
\end{array} \begin{array}{c}
\mapsto \{(x, *) \mid x \in k\}
\end{array}$$

defines a unique prop morphism $S: \mathbb{I}R \to \text{LinRel}_k$ that preserves $(-)^\dagger$. For instance,

$$S(\mapsto) = S(\mapsto \dagger) = (S(\mapsto \circ)) = \{(\ast, x) \mid x \in k\}.$$
Theorem 8 ([7]). \( S : \mathbb{IR} \to \text{LinRel}_k \) is an isomorphism.

Let us explain the relationship of \( S \) with \( S' \) from Example 6. Observe that any \( A : m \to n \) in \( \mathbb{IR} \) built out of the leftmost five generators of Fig. 1 (drawn \( \begin{array}{c} \text{m} & \text{n} \\ \text{A} & \end{array} \)) is also in \( \mathbb{AR} \) and, similarly, any term built of the five rightmost generators (\( \begin{array}{c} \text{m} & \text{n} \\ \text{A} & \end{array} \)) is in \( \mathbb{AR}^{op} \). Indeed, we have prop embeddings \( \mathbb{AR} \to \mathbb{IR} \leftarrow \mathbb{AR}^{op} \). Similarly, there are embeddings \( \text{Mat}_{\mathbb{R}} \to \text{LinRel}_k \leftarrow \text{Mat}_{\mathbb{R}}^{op} \) mapping a matrix to its graph, and the following commutes [7]:

\[
\begin{array}{ccc}
\mathbb{AR} & \longrightarrow & \mathbb{IR} \\
\downarrow S' & & \downarrow S \\
\text{Mat}_{\mathbb{R}} & \longrightarrow & \text{LinRel}_k \\
& \uparrow & \\
& \text{Mat}_{\mathbb{R}}^{op} & \leftarrow \text{Mat}_{\mathbb{R}}^{op}
\end{array}
\]

The following result informs us that every arrow of \( \mathbb{IR} \) can be written in span form.

Lemma 9 ([7]). For all \( \begin{array}{c} \text{m} \to \text{n} \\ \text{A} & \end{array} \) in \( \mathbb{IR} \) there exist \( k \in \mathbb{N} \), \( \begin{array}{c} \text{m} \times k \\ \text{A}_1 & \end{array} \) and \( \begin{array}{c} \text{n} \times k \\ \text{A}_2 & \end{array} \) such that \( \begin{array}{c} \text{m} \to \text{n} \\ \text{A} & \end{array} = \begin{array}{c} \text{m} \to \text{k} \\ \text{A}_1 & \end{array} \) \( \begin{array}{c} \text{k} \to \text{n} \\ \text{A}_2 & \end{array} \). Moreover, the following property of \( \mathbb{AR} \) also holds in \( \mathbb{IR} \).

Lemma 10 ([7]). For all \( \begin{array}{c} \text{m} \to \text{n} \\ \text{A} & \end{array} \), \( \begin{array}{c} \text{m} \times \text{k} \\ \text{A}_1 & \end{array} \) and \( \begin{array}{c} \text{n} \times \text{k} \\ \text{A}_2 & \end{array} \) such that \( \begin{array}{c} \text{m} \to \text{n} \\ \text{A} & \end{array} = \begin{array}{c} \text{m} \times \text{k} \\ \text{A}_1 & \end{array} \) \( \begin{array}{c} \text{k} \to \text{n} \\ \text{A}_2 & \end{array} \). The equivalence classes are the arrows of a 2-category \( T(\Sigma, I) \) that we call an ordered prop.

5 Symmetric Monoidal Inequality Theories and Ordered Props

We reviewed the construction of props from SMTs in the previous section. In order to capture inequalities of terms, however, we need a new notion. We thus introduce the concept of a Symmetric Monoidal Inequality Theory (SMIT), which allows the specification of a partial order on terms built out of generators, analogously to how SMTs specify equivalence relations.

Definition 11 (Symmetric Monoidal Inequality Theory). A SMIT is a pair \((\Sigma, I)\). As for SMTs, \( \Sigma \) is a collection of generators \( o : m \to n \), and \( I \) is a set of pairs \((t_1, t_2)\) of \( \Sigma \)-terms with the same (co)arity, but we now think of them as representing inequalities. That is, where before the interpretation of a pair \((t_1, t_2)\) was that \( t_1 = t_2 \), we now stipulate that \( t_1 \leq t_2 \).

Set \( I \) leads to a preorder on terms by reflexive and transitive closure. A partial order arises through anti-symmetry: \( t_1 \) and \( t_2 \) are equated when \( t_1 \leq t_2 \) and \( t_2 \leq t_1 \). We will use \( \leq \), or \( \leq I \) when \( I \) is clear from context, and write the corresponding equivalence as equality. The equivalence classes are the arrows of a 2-category \( T(\Sigma, I) \) that we call an ordered prop.

Definition 12 (Ordered Prop). A 2-prop is a strict symmetric monoidal 2-category whose objects are natural numbers and monoidal product on objects is addition. An ordered prop \( T \) is a 2-prop which is locally posetal, that is, where every hom-category is a poset – i.e. there is at most one 2-cell \( \leq \) between any two arrows. Together with ordered prop morphisms (identity-on-objects strict monoidal 2-functors) we have a category \( \text{OrdPROP} \).

Since ordered props are a kind of 2-category, in any ordered prop, for all \( f, f', g, g' \), we have:

\[
\text{if } f \leq f' \text{ and } g \leq g' \text{ then } f \circ g \leq f' \circ g'.
\] (16)

\[
\text{if } f \leq f' \text{ and } g \leq g' \text{ then } f \oplus g \leq f' \oplus g'.
\] (17)

Example 13. The prop \( \text{LinRel}_k \) of linear relations has a partial order on arrows given by inclusion as subspaces. It is straightforward to check (16) and (17). The prop \( \text{Mat}_{\mathbb{R}} \) of matrices (Example 6) can be also regarded as an ordered prop with discrete order.
Returning to SMITs, the arrows of \( T(\Sigma, I) \) are the equivalence classes of \( \Sigma \)-terms; since arities and coarities are respected, we may use the partial order \( \leq I \) to define the 2-cells in the hom-category \( T(\Sigma, I)(m, n) \). It follows that \( T(\Sigma, I) \) is an ordered prop. Note that SMITs are a generalisation of SMTs. First, any prop can be made into a (discrete) ordered prop by adding identity 2-cells. This gives an embedding (a faithful homomorphism of ordered pros: a strict identity-on-objects symmetric monoidal 2-functor) \( \text{PROP} \to \text{OrdPROP} \).

\[
\begin{align*}
\text{Remark 14.} & \quad \text{Any SMT } (\Sigma, E) \text{ can be considered as a SMIT by taking the symmetric closure of } E: i.e. I = E \cup E^\circ. \text{ Then the image of the prop generated by the SMT } (\Sigma, E) \text{ under the embedding is isomorphic to the ordered prop given by SMIT } (\Sigma, I).
\end{align*}
\]

\section{Presenting the 2-dimensional structure of \( \text{LinRel}_k \)}

In this section we prove our main theorem. We extend the SMT of Interacting Hopf Algebras to a SMIT and the isomorphism of Theorem 8 to a 2-isomorphism of ordered props. In other words, we characterise the subset order of linear relations (Example 13).

The symmetry discussed in Section 4 is broken in the ordered setting. Indeed, to get from the SMT to a SMIT we follow the procedure of Remark 14 and add just one inequality (2): \( \leq \). Interpreted as linear relations (via \( S : \text{IH}R \to \text{LinRel}_k \)), (2) says that the unique 0-dimensional subspace \( \{0\} \) of \( k \) considered as a \( k \)-vector space is included in the unique 1-dimensional subspace, i.e. \( k \) itself. This is, of course, a strict inclusion.

\[
\begin{align*}
\text{Theorem 15.} & \quad \text{IH}R \cong \text{LinRel}_k \text{ as ordered props. }
\end{align*}
\]

For the proof we need to recall some elementary linear algebra. Regarding an \( m \times n \) matrix \( A \) as a list of its column vectors \( a_1, a_2, \ldots, a_n \), the span of \( A \) \( (\text{Sp}(A)) \) is the linear subspace of \( k^m \) with elements linear combinations \( \lambda_1 a_1 + \lambda_2 a_2 + \cdots + \lambda_n a_n, \lambda_i \in k \). The following is a well-known fact of linear algebra (see, e.g. [2, Proposition 2.13]).

\[
\begin{align*}
\text{Lemma 16.} & \quad \text{Suppose that for some } m \times n \text{ matrix } A \text{ we have } \text{Sp}(A) \subseteq V. \text{ Then there exists } m \times n' \text{ matrix } C \text{ such that } V = \text{Sp}(A, C).
\end{align*}
\]

\[
\begin{align*}
\text{Proof of Theorem 15.} & \quad \text{Inequation (2) is clearly sound, we thus only have to show completeness; that is (2) suffices to account for any inclusion between arbitrary linear relations.}
\end{align*}
\]

Let therefore \( A, B: m \to n \) be arrows of \( \text{IH}R \) such that \( S(A) \subseteq S(B) \subseteq k^m \times k^n \). Now \( k^m \times k^n \cong k^{m+n} \times k^0 \cong k^{m+n} \); this, diagrammatically, means the following manipulation:

\[
\begin{align*}
\text{Using this “rewiring” argument we may assume w.l.o.g. that } A, B : m \to 0. \text{ Further, using Lemma 9, we may assume that } A, B \text{ consist only of the rightmost five generators in Figure 1: indeed, } - \to \to = - \to \to \to = - \to \to \to \text{ by Lemmas 9 and 10. It is thus harmless to consider } A \text{ and } B \text{ as matrices, and our initial assumption means that } \text{Sp}(A) \subseteq \text{Sp}(B). \text{ By the conclusion of Lemma 16, there exists } C \text{ such that } \text{Sp}(B) = \text{Sp}(A, C). \text{ Diagrammatically (via } S), \text{ this gives the following, where for readability we omit decorating the wires:}
\end{align*}
\]

\[
\begin{align*}
\text{But we have}
\end{align*}
\]

\[
\begin{align*}
\text{showing that } A \leq B \text{ is derivable from (2).}
\end{align*}
\]
While we have shown that (2) suffices to characterise inclusions between subspaces, it is convenient to identify some structural properties that our inequational theory satisfies. By doing so, we are building up a toolbox—useful for reasoning in applications—of principles for reasoning about the structure of the order between linear relations.

Below we use the notion of adjunction in an ordered prop: arrow \( f : m \to n \) has a right adjoint if there exists \( g : n \to m \) such that \( \text{id}_m \leq f \circ g \) and \( g \circ f \leq \text{id}_n \), in which case we write \( f \dashv g \). Right adjoints, if they exist, are unique: if also \( f \dashv g' \) then \( g = g' \).

**Definition 17.** An abelian bicategory \([11]\) \( A \) is a (loc. posetal) monoidal bicategory where:

(i) every object \( a \) is a commutative comonoid \((\xrightarrow{a} a, a)\) with right adjoints \( \xleftarrow{a} a \), and a commutative monoid \((\xrightarrow{a} a, a)\) with right adjoints \( \xleftarrow{a} a \).

(ii) \((\xrightarrow{a} a, a)\) and \((\xrightarrow{a} a, a)\) with their right adjoints satisfy the Frobenius equations:

\[
\begin{align*}
\xrightarrow{a} a & \leq \xrightarrow{a} a, \\
\xleftarrow{a} a & \leq \xleftarrow{a} a, \\
\text{id}_I & \leq \xrightarrow{a} a, \\
\xleftarrow{a} a & \leq \text{id}_I.
\end{align*}
\]

(iii) every arrow \( \xrightarrow{a} a \) is a lax \((\xrightarrow{a} a, \xleftarrow{a} a)\)-comonoid homomorphism and a lax \((\xrightarrow{a} a, \xleftarrow{a} a)\)-monoid homomorphism:

\[
\begin{align*}
\xrightarrow{a} a & \leq \xrightarrow{a} a, \\
\xleftarrow{a} a & \leq \xleftarrow{a} a.
\end{align*}
\]

A more concise definition is: \( A \) and \( A^{\text{op}} \) are both bicategories of relations in the sense of \([11]\).

Below, we show that, as an ordered prop, \( \mathbb{H}_R \) is an abelian bicategory. For each object \( n \in \mathbb{N} \), comonoids and monoid structures are defined inductively as in Remark 7. A straightforward induction generalises the Frobenius equations for all \( n \) in (19), given that they are present for \( n = 1 \) in Fig. 1. Next we tackle the case of the black units (the rightmost two black inequations of (18)). The unit of the adjunction is a 2-cell witnessing \( \text{id}_0 \leq \bullet \bullet \) and these terms are equated in Fig. 1. It remains to show existence of the counit. For \( n = 1 \):

\[
\xrightarrow{a} a \leq \xrightarrow{a} a = \xleftarrow{a} a = \bullet \bullet
\]

The above argument easily generalises to all \( n \).

Showing adjointness for the black comultiplication (the leftmost two black inequations of (18)) amounts to demonstrating that \( \bullet \bullet \leq \bullet \bullet \) and \( \bullet \bullet \leq \bullet \bullet \). The second is the black special equation in Fig. 1. The first follows from the adjointness of the unit and counit:

\[
\bullet \bullet \leq \bullet \bullet = \bullet \bullet = \bullet \bullet
\]

For the white case in (18), the inequations are opposite. The same proofs with colours and the sense of the inequality exchanged give the results that \( \circ \circ \leq \bullet \bullet \) and \( \bullet \bullet \leq \circ \circ \).
Now, to show that all arrows are lax comonoid homomorphisms, it is enough to check that each of the generators obeys the conditions of (20). Several of these are in fact equalities. The derivations for the two interesting cases are given below:

\[
\begin{align*}
\bullet &\leq \quad \bullet \\
\implies &\leq \\
\implies &\leq \quad \bullet
\end{align*}
\]
Again, the white case (21) is symmetric.

7 The 2-dimensional algebra of Linear Relations

The structure identified in the previous sections enables us to highlight some interesting properties of the ordering \(\leq\). We start with a few elementary, but useful observations.

\[\text{Lemma 18.} \quad \text{For all } A, B \in \mathbb{IR}(m, n), \text{ the following hold:}\]

(a) If \(A \leq B\), then \(A^\dagger \leq B^\dagger\);
(b) \((A^\dagger)^\circ = (A^\circ)^\delta\);
(c) If \(A \leq B\), then \(A^\circ \geq B^\circ\);
(d) \((A^\dagger)^\circ = (A^\circ)^\dagger\).

\[\text{Proof.} \quad \text{Trivial by unfolding the definitions.} \]

\[\text{Lemma 19.} \quad \text{For all } A, B \in \mathbb{IR}(m, n), \text{ the following hold:}\]

(a) \((A \lor B)^\dagger = A^\dagger \lor B^\dagger\) and \(\bot^\dagger = \bot\);
(b) \((A \land B)^\circ = A^\circ \land B^\circ\) and \(\top^\circ = \top\);
(c) \((A \lor B)^\circ = A^\circ \lor B^\circ\) and \(\bot^\circ = \top\);
(d) \((A \land B)^\circ = A^\circ \lor B^\circ\) and \(\bot^\circ = \top\).

\[\text{Proof.} \quad \text{Trivial by unfolding the definitions.} \]

\[\text{Theorem 20.} \quad \text{The operations } (\lor, \land, \tau, \top) \text{ define a lattice structure on every homset } \mathbb{IR}(m, n) \text{ wrt the ordering } \leq.\]

\[\text{Proof.} \quad \text{First observe that for all } A: m \to n, \text{ we have, by the rightmost inequations in (18) and (20), that } A \leq A; \tau \leq \tau. \text{ By Lemmas 18(b) and 19(d), one deduces } A \geq \perp. \]

Now, it is enough to check that both \((\mathbb{IR}(m, n), \land, \tau)\) and \((\mathbb{IR}(m, n), \lor, \bot)\) are commutative and idempotent monoids. Observe that if this holds for \((\mathbb{IR}(m, n), \land, \tau)\) then, by Lemmas 18(b) and 19(d), it holds also for \((\mathbb{IR}(m, n), \lor, \bot)\). The fact that \((\mathbb{IR}(m, n), \land, \tau)\) is a commutative monoid follows immediately from the fact that \((m, m, m)\) and \((m, m, m)\) are commutative (co)monoids. For idempotency we observe the following chain of inequalities.

\[
\begin{align*}
\text{A} &\leq \text{A} \\
\text{A} &\leq \text{A} \\
\text{A} &\leq \text{A}
\end{align*}
\]
Refinement for Signal Flow Graphs

\[ \begin{align*}
\text{Refinement for Signal Flow Graphs:} & \quad \text{(1,2)} \quad \text{(1,0)} \\
\text{c: (m, z)} & \quad \text{d: (z, n)} \\
\text{e: (1,1)} & \quad \text{d: (m, n)} \\
\text{c: (0,1)} & \quad \text{Tr(c): (m, n)}
\end{align*} \]

\[ \begin{align*}
\text{Figure 2 Sort inference rules.} & \quad \text{\((k \rightarrow k) \rightarrow k\)} \\
\text{\((k \rightarrow k) \rightarrow k\)} & \quad \text{\((k \rightarrow k) \rightarrow k\)} \\
\text{\((k \rightarrow k) \rightarrow k\)} & \quad \text{\((k \rightarrow k) \rightarrow k\)}
\end{align*} \]

\[ \begin{align*}
\text{\(s\) \xrightarrow{k} \(s'\)} & \quad \text{\(t\) \xrightarrow{w} \(t'\)} \\
\text{\(s\) \xrightarrow{u_1} \(s'\)} & \quad \text{\(s\) \xrightarrow{u_2} \(s'\)} \\
\text{\(s\) \xrightarrow{u_1 + u_2} \(s'\)} & \quad \text{\(s\) \xrightarrow{u_1} \(s'\)} \\
\text{\(s\) \xrightarrow{u_2} \(s'\)} & \quad \text{\(s\) \xrightarrow{u_1 + u_2} \(s'\)} \\
\text{\(\text{Tr}^k(s)\)} & \quad \text{\(\text{Tr}^k(s)\)}
\end{align*} \]

\[ \begin{align*}
\text{Figure 3 Structural rules for operational semantics, with \(k, l\) ranging over \(k\) and \(u, v, w\) vectors of elements of \(k\) of the appropriate size.}
\end{align*} \]

Now to see that \(\wedge\) defines a meet, note that \(A \wedge B \leq A\) and, by a symmetric argument, \(A \wedge B \leq B\). Assuming that \(A \leq C\) and \(B \leq C\) we have

\[ \begin{align*}
A\wedge B & \leq C \\
A\wedge B & \leq C
\end{align*} \]

The argument showing that \(\vee\) defines a join is again symmetric.

\[ \text{Lemma 21. For all } A, B, C \in \mathbb{H}_R(m, n), \text{ the following hold:} \]

\[ \begin{align*}
\text{(a) } & \quad C : (A \wedge B) \leq (C : A) \wedge (C : B) \text{ and } C : \top \leq \top; \\
\text{(b) } & \quad (A \wedge B) ; C \leq (A ; C) \wedge (B ; C) \text{ and } \top ; C \leq \top; \\
\text{(c) } & \quad (A \vee B) ; C \geq (A ; C) \vee (B ; C) \text{ and } \bot ; C \geq \bot; \\
\text{(d) } & \quad (A \vee B) \geq (C ; A) \vee (C ; B) \text{ and } C ; \bot \leq \bot.
\end{align*} \]

Proof. Part (a) follows from (20). With (a), Lemmas 18(a) and 19(b) imply (b). With (b), Lemmas 18(b) and 19(d) imply (c). With (c), Lemmas 18(a) and 19(a) imply (d).

To summarise, we have a well-behaved set of operations

\[ \top, \bot, \land, \lor, (\neg)^\dagger, (\neg)^0, ;, \text{id} \]

which, because of its similarity to the algebra of relations [14], we call the algebra of linear relations. Observe that the operations \(\top, \land, (\neg)^\dagger, ;\) and \(\text{id}\) have exactly the same meaning: full relation, intersection, inverse, composition and identity relation. Instead \(\bot, \lor, (\neg)^0\) which in the algebra of relations denote, respectively, empty relation, union and complement, do not coincide. The reason is that, in general, these operations cannot be defined on linear relations: e.g., the union of two linear relations may not be linear.
8 Back to Signal Flow Graphs

We recall from [6] the Directed Signal Flow Calculus. The syntax is given by the grammar below where $k$ ranges over a fixed field $k$.

$$c ::= \text{...}$$

A sort is a pair $(m, n)$, with $m, n \in \mathbb{N}$. We shall consider only terms that are sortable, according to the rules of Fig. 2. An inductive argument confirms uniqueness of sorts: if $c : (m, n)$ and $c : (m', n')$ then $m = m'$ and $n = n'$. We will refer to sortable terms as circuits since, intuitively, a term $c : (m, n)$ represents a circuit with $m$ inputs on the left and $n$ outputs on the right.

In the syntax specification we used a graphical rendering of the components: we will seldom write terms in the traditional way and instead represent them as diagrams:

- $c; c'$ is drawn
- $c \oplus c'$ is drawn
- $\text{Tr}(c)$ is drawn

The graphical notation identifies some syntactically different terms, e.g. $(c_1 \oplus c_2) : (d_1 \oplus d_2)$ and $(c_1 ; d_1) \oplus (c_2 ; d_2)$. This is harmless (see Remark 1 in [6]) since such circuits are observationally equivalent wrt the operational semantics that we introduce next.

The operational semantics interprets terms as stream transducers. The wires carry elements of a field $k$ that enter and exit through input and output ports. Formally, it is a transition system that has augmented circuits as states: each delay component and each guarded feedback (Tr) are assigned some value $k \in k$. States are obtained by replacing delays and feedbacks in the syntax specification with $\rightarrow_{d,k}$ and $\text{Tr}^k$ for each $k \in k$. We only consider sortable states; the discipline is obtained by adding the following to Fig. 2.

$$\rightarrow_{d,k} : (1, 1) \quad \text{and} \quad \frac{c ; (1+m, 1+n)}{\text{Tr}^k(c) : (m, n)}$$

The structural rules for operational semantics are given in Fig. 3 where we use strings of length $n$ to represent vectors in $k^n$. If state $s : (m, n)$ is the source of a transition $\overset{s}{\rightarrow} t$ then $t$ is also a state with sort $(m, n)$ and $v$ and $w$ are strings (vectors) in $k^n$ and $k^n$, respectively. Intuitively, $s \overset{s}{\rightarrow} t$ means that $s$ can become $t$ in one step whenever it inputs $v$ on the $m$ ports on the left and outputs $w$ on the $n$ ports on the right. Each circuit $c$ then yields a transition system with a chosen initial state $s_0$ of $c$, obtained by replacing delays and feedbacks in $c$ with $\rightarrow_{d,0}$ and $\text{Tr}^0$: this means that we only consider executions where the registers are initialised with 0. A different semantics is considered in [13] where registers can be initialised with arbitrary values.

A computation of a circuit $c$, is an infinite path $s_0 \overset{\alpha_0}{\rightarrow} s_1 \overset{\alpha_1}{\rightarrow} \ldots$ in the transition system of $c$, starting from its initial state $s_0$. When $c$ has sort $(m, n)$, each $u_i$ and $v_i$ are strings over $k$, say $k_{i1} \ldots k_{im}$ and $l_{i1} \ldots l_{in}$, respectively. The trace, also called trajectory, of this computation is then a pair of vectors $\alpha = \left( \begin{array}{c} \alpha_1 \\ \vdots \\ \alpha_n \end{array} \right), \beta = \left( \begin{array}{c} \beta_1 \\ \vdots \\ \beta_m \end{array} \right)$ where $\alpha_j = k_{ij}k_{ijj} \ldots$ and $\beta_j = l_{ij}l_{ijj} \ldots$. For example, consider the $(1,1)$ circuit in (3): the first three steps of an infinite computation are illustrated in (4). The trace for this computation is thus $(1; 0; 0; \ldots, 1; 2; 3; 5; \ldots)$.

We write $it(c)$ for the set of traces, and this is our notion of observable behaviour. Two circuits $c$ and $d$ are observationally equivalent, written $c \sim d$, iff $it(c) = it(d)$.  

---

Note: The above text is a natural representation of the document content, focusing on the operational semantics and the directed signal flow calculus, without the need for any visual aids like diagrams or tables.
A few considerations are in order about the role of the denotational semantics given in [6]. The signal flow calculus is canonical in the sense that it enjoys a Kleene-like theorem [8, Theorem 7.4]: it denotes all and only the rational functions on streams (see [18] and [17]). Moreover the denotational semantics is fully abstract with respect to the observational equivalence (Corollary 2 and Proposition 4 in [6]) and, from this correspondence, a sound and complete axiomatisation for \( \sim \) follows. We focus on some technical details of this axiomatisation below.

The idea is to translate circuits of sort \((m,n)\) into arrows \(m \to n\) of \(\mathbb{H}_k[x]\), where \(k[x]\) is the principal ideal domain of polynomials with indeterminate \(x\) and coefficients from \(k\). Intuitively, the inductively defined translation \(\mathcal{E}\) “erases directions” from the wires:

\[
\begin{align*}
\begin{array}{c}
\begin{array}{c}
\bullet \\
\circ
\end{array}
\end{array}
\mapsto
\begin{array}{c}
\begin{array}{c}
\bullet \\
\circ
\end{array}
\end{array}, \quad
\begin{array}{c}
\begin{array}{c}
\circ \\
\circ
\end{array}
\end{array}
\mapsto
\begin{array}{c}
\begin{array}{c}
\bullet \\
\circ
\end{array}
\end{array}, \quad
\begin{array}{c}
\begin{array}{c}
\circ
\end{array}
\end{array}
\mapsto
\begin{array}{c}
\begin{array}{c}
\bullet
\end{array}
\end{array}, \quad
\begin{array}{c}
\begin{array}{c}
\circ
\end{array}
\end{array}
\mapsto
\begin{array}{c}
\begin{array}{c}
\bullet
\end{array}
\end{array}, \quad
\begin{array}{c}
\begin{array}{c}
\circ
\end{array}
\end{array}
\mapsto
\begin{array}{c}
\begin{array}{c}
\circ
\end{array}
\end{array}, \quad
\begin{array}{c}
\begin{array}{c}
\circ
\end{array}
\end{array}
\mapsto
\begin{array}{c}
\begin{array}{c}
\circ
\end{array}
\end{array}, \quad
\begin{array}{c}
\begin{array}{c}
\circ
\end{array}
\end{array}
\mapsto
\begin{array}{c}
\begin{array}{c}
\circ
\end{array}
\end{array}
\end{array}
\end{align*}
\]

where \(\begin{array}{c}
\begin{array}{c}
\circ
\end{array}
\end{array}\) and \(\begin{array}{c}
\begin{array}{c}
\circ
\end{array}
\end{array}\), in \(\mathbb{H}_k[x]\), correspond to polynomials \(k\) and \(x\) in \(k[x]\).

For an example consider the circuits \(\begin{array}{c}
\begin{array}{c}
\circ
\end{array}\end{array}\) and \(\begin{array}{c}
\begin{array}{c}
\circ
\end{array}\end{array}\) in (3) and (5): the corresponding string diagrams \(\mathcal{E}(\begin{array}{c}
\begin{array}{c}
\circ
\end{array}\end{array})\) and \(\mathcal{E}(\begin{array}{c}
\begin{array}{c}
\circ
\end{array}\end{array})\) are shown in (6).

The following ensures that the theory of Fig. 1 is sound and complete for trace equivalence.

**Theorem 22** ([6]). \(c \sim d\) iff \(\mathcal{E}(c) = \mathcal{E}(d)\), for all circuits \(c, d\).

To prove equivalence of signal flow calculus terms it is thus enough to view them as string diagrams in \(\mathbb{H}_k[x]\) by forgetting flow direction, and use the equational theory of Fig. 1.

The reader may wonder why we introduced the directed signal flow calculus rather than using string diagrams directly. The reason is that string diagrams of \(\mathbb{H}_k[x]\) are undirected and flow directionality is essential to execute them (see Remark 2 in [6]). String diagrams, however, do provide a useful language to reason about signal flow graphs. For instance, using the algebra of linear relations from Section 7, the opposite of an arbitrary circuit \(c\) can be specified by the string diagram \(\mathcal{E}(c)\). It is therefore natural to think of string diagrams in \(\mathbb{H}_k[x]\) as specifications and of circuits in the directed signal flow calculus as implementations. More formally, we say that a specification \(A\) (an arrow of \(\mathbb{H}_k[x]\)) refines a specification \(B\) whenever \(A \leq B\) and we say that a circuit \(c\) implements a specification \(A\) whenever \(\mathcal{E}(c)\) refines \(A\), i.e., \(\mathcal{E}(c) \leq A\).

**Remark 23.** One could have defined \(c \preceq d\) if \(\mathcal{E}(c) \subseteq \mathcal{E}(d)\) but this notion would collapse to \(\sim\), since the observational behaviour of any circuit, which we have defined as a relation, is actually the graph of a function. To see this, note that the operational semantics of Fig. 3 is deterministic: given any state \(s\) and transitions \(s \xrightarrow{\nu} s'\) and \(s \xrightarrow{\nu'} s''\), it follows that \(\nu = \nu'\). A similar, but non-deterministic, semantics subsuming that of Fig. 3 was given in [6, Fig. 2] for arbitrary string diagrams. In fact, losing direction of signal flow makes the definition simpler, since the feedback becomes expressible in terms of the more basic components and does not thus need a separate structural rule. It is the possibly non-deterministic nature of string-diagrams-as-SFG-specifications that makes the refinement relation interesting.

We now return to the motivating example of Section 2. The fact that the circuit in (5) solves the sustainable rabbit farming problem is witnessed by the fact that it is an implementation of \(\mathcal{E}(\begin{array}{c}
\begin{array}{c}
\circ
\end{array}\end{array})\). Here, since the behaviour of \(\begin{array}{c}
\begin{array}{c}
\circ
\end{array}\end{array}\) is invertible,
there is an equivalence: see the derivation in (7). Instead, the sustainable farming problem for rabbits and guinea pigs cannot be solved by equational reasoning since the combined SFG (8(ii)) (henceforth $\text{comb}$) is not invertible. To prove that SFG (8(iv)) (henceforth $\text{sol}$) is a solution, we should show that it implements $E(\text{comb})$, namely we should check that $E(\text{comb}) \leq E(\text{comb})$. It follows from the general fact shown below; taking $\lambda = \frac{1}{2}$ gives the claimed solution.

\[
\begin{array}{c}
\lambda \\
\frac{1}{1-\lambda} \\
\end{array}
\leq
\begin{array}{c}
\lambda \\
\frac{1}{1-\lambda} \\
\end{array}
= -D \rho = -\rho
\]

9 Related work

Although we concentrated on the discrete semantics, signal flow graphs also have a continuous incarnation where delays act as integrators; for this reason they are a useful foundational model in signal processing and control theory: as a consequence, for computer scientists they are also important as models of cyber-physical systems that can be analysed and verified in concert with discrete models. For example, in loc. cit. the authors study SFGs with the aid of block diagrams that are closely related to the Signal Flow Calculus of Section 8.

The operation (22) of passing from the directed calculus to string diagrams by “erasing arrowheads” is similar in spirit to the ideas of Willems [23], who argued that concepts of input and output are inherently non-compositional, complicate the mathematics, and—perhaps most importantly—do not actually exist in the underlying physical reality. It is this realisation that gives rise to the equational theory of Interacting Hopf Algebras. Moreover, Baez and Erbele [3] prove that the same equational theory is suitable for the continuous behaviour. Remarkably similar symmetric monoidal theories appear in concurrency [9, 10, 21] and quantum computing [12]. None of these works, however, investigates the underlying posetal structure. We believe that the structure of cartesian and abelian bicategories [11] may be successfully exploited in those fields.

References

Refinement for Signal Flow Graphs

Brzozowski Goes Concurrent – A Kleene Theorem for Pomset Languages

Tobias Kappé\textsuperscript{1}, Paul Brunet\textsuperscript{2}, Bas Luttik\textsuperscript{3}, Alexandra Silva\textsuperscript{4}, and Fabio Zanasi\textsuperscript{5}

\textsuperscript{1} University College London, London, United Kingdom
\texttt{tkappe@cs.ucl.ac.uk}
\textsuperscript{2} University College London, London, United Kingdom
\textsuperscript{3} Eindhoven University of Technology, Eindhoven, The Netherlands
\textsuperscript{4} University College London, London, United Kingdom
\textsuperscript{5} University College London, London, United Kingdom

Abstract

Concurrent Kleene Algebra (CKA) is a mathematical formalism to study programs that exhibit concurrent behaviour. As with previous extensions of Kleene Algebra, characterizing the free model is crucial in order to develop the foundations of the theory and potential applications. For CKA, this has been an open question for a few years and this paper makes an important step towards an answer. We present a new automaton model and a Kleene-like theorem that relates a relaxed version of CKA to series-parallel pomset languages, which are a natural candidate for the free model. There are two substantial differences with previous work: from expressions to automata, we use Brzozowski derivatives, which enable a direct construction of the automaton; from automata to expressions, we provide a syntactic characterization of the automata that denote valid CKA behaviours.

1998 ACM Subject Classification D.1.3 Parallel Programming, F.1.1 Models of Computation, F.1.2 Modes of Computation, F.4.3 Formal Languages

Keywords and phrases Kleene theorem, Series-rational expressions, Automata, Brzozowski derivatives, Concurrency, Pomsets

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2017.25

1 Introduction

In their CONCUR’09 paper [5], Hoare, Möller, Struth, and Wehrman introduced Concurrent Kleene Algebra (CKA) as a suitable mathematical framework to study concurrent programs, in the hope of achieving the same elegance that Kozen did when using Kleene Algebra (and extensions) to provide a verification platform for sequential programs.

CKA is a seemingly simple extension of Kleene Algebra (KA): it adds a parallel operator that allows to specify concurrent behaviours compositionally. However, extending the existing KA toolkit – importantly, completeness and decidability results – turns out to be challenging. A fundamental missing ingredient is a characterization of the free model for CKA. This is in striking contrast with KA, where these topics are well understood. Several authors [6, 8] have conjectured the free model to be series-parallel pomset languages – a generalization of regular languages to sets of partially ordered words.

\textsuperscript{*} This work was partially supported by the ERC Starting Grant ProFoundNet (grant code 679127).
In KA, Kleene’s theorem provided a pillar for developing the toolkit and axiomatization [13], and, by extension, characterizing the free model. In this light, we pursue a Kleene Theorem for CKA. Specifically, we study series-rational expressions, with a denotational model in terms of pomset languages. Our main contribution is a Kleene Theorem for series-rational expressions, based on constructions faithfully translating between the denotational model and a newly defined operational model, which we call pomset automata. In a nutshell, these are finite-state automata in which computations from a certain state may branch into parallel threads that contribute to the language of s whenever they both reach a final state.

We are not the first to attempt such a Kleene theorem. However, earlier works [16, 8] fall short of giving a precise correspondence between the denotational and operational models, due to the lack of a suitable automata restriction ensuring that only valid behaviours are accepted. We overcome this situation by introducing a generalization of Brzozowski derivatives [3] in the translation from expressions to automata. This guides us to a syntactic restriction on automata (rather than the semantic condition put forward in previous works), which guarantees the existence of a reverse construction, from automata to expressions. Moreover, following the Brzozowski route allows us to bypass a Thompson-like construction [19], avoiding the introduction of ϵ-transitions and non-determinism present in the aforementioned works.

Since series-parallel expressions do not include the parallel analogue of the Kleene star (the “parallel star”), and our denotational model is not sound for the exchange law (which governs the interaction between sequential and parallel composition), our contribution is most accurately described as an operational model for weak Bi-Kleene Algebra. We leave it to future work to extend our construction to work with a denotational model that is sound for the exchange law (thus moving to weak Concurrent Kleene Algebra), as well as add the parallel star operator (arriving at Concurrent Kleene Algebra proper).

The remainder of this paper is organized as follows. In Section 2, we introduce the necessary notation. In Section 3, we introduce our automaton model as well as some notable subclasses of automata. In Section 4, we discuss how to translate a series-rational expression to a semantically equivalent pomset automaton, while in Section 5 we show how to translate a suitably restricted class of pomset automata to series-rational expressions. We contrast results with earlier work in Section 6. Directions for further work in are listed in Section 7.

To save space, proofs of lemmas are omitted from this paper. For a discussion that includes a proof for every lemma, we refer to the extended version of this paper [9].

## 2 Preliminaries

Let $S$ be a set; we write $2^S$ for the set of all subsets of $S$, and $\binom{S}{2}$ for the set of multisets over $S$ of size two. An element of $\binom{S}{2}$ containing $s_1, s_2 \in S$ is written $\{s_1, s_2\}$; note that $\{s_1, s_2\} = \{s_2, s_1\}$, and that $s_1$ may be the same as $s_2$. We use the symbols $\phi$ and $\psi$ to denote multisets. If $S$ and $I$ are sets, and for every $i \in I$ there exists an $s_i \in S$, we call $(s_i)_{i \in I}$ an $I$-indexed family over $S$. We say that a relation $\prec \subseteq S \times S$ is a strict order on $S$ if it is irreflexive and transitive. We refer to $\prec$ as well-founded if there are no infinite descending $\prec$-chains, i.e., no family $(s_n)_{n \in \mathbb{N}}$ over $S$ such that $\forall n \in \mathbb{N}, s_{n+1} \prec s_n$. Throughout the paper we fix a finite set $\Sigma$ called the alphabet, whose elements are symbols usually denoted with $a$ and $b$. Lastly, if $\rightarrow \subseteq X \times Y \times Z$ is a ternary relation, we write $x \xrightarrow{y} z$ instead of $\langle x, y, z \rangle \in \rightarrow$. 
Partially-ordered multisets

that are not related by example, describing the pomset forces the events in the left pomset to be ordered before those in the right pomset. An

Note that

function from

\( V \) and \( U \) ▶

are an antichain of size \( U \) in ▶

that whenever we have two pomsets, we pick representatives that have disjoint carrier sets. A little importance; it is the order and labelling that matters. For this reason, we tacitly assume being isomorphism classes, the content of the carrier of the chosen representative is of very

in its isomorphism class. We use the symbols

is called
to refer to the pomset with a single point labelled \( a \).

For instance, suppose a recipe for caramel-glazed cookies tells us to

prepare
glaze

caramelize
dough,

bake

glaze

caramelize

cookies in the oven, and

caramelize

glaze

the finished cookies. Here,

prepare
and caramelize, while glaze succeeds both. A pomset representing this process could be \(( C, \leq_C, \lambda_C)\), where \( C = \{ a, b, c \} \) and \( \leq_C \) is such that \( a \leq_C b \leq_C d \) and \( a \leq_C c \leq_C d \), and \( \lambda_C(a) = \text{prepare}, \lambda_C(b) = \text{bake}, \lambda_C(c) = \text{caramelize}, \lambda_C(d) = \text{glaze} \).

Note that words are just finite pomsets with a total order. We will sometimes use \( a \in \Sigma \) to refer to the pomset with a single point labelled \( a \) (and the obvious order); such a pomset is called primitive. A pomset can be represented as a Hasse diagram, where nodes have labels in \( \Sigma \). For instance, the Hasse diagram for the pomset \( C \) above is drawn in Figure 1a.

To simplify notation, we refer to a pomset by the carrier \( U \) of a labelled poset \( (U, \leq_U, \lambda_U) \) in its isomorphism class. We use the symbols \( U, V, W \) and \( X \) to denote pomsets. Pomsets being isomorphism classes, the content of the carrier of the chosen representative is of very little importance; it is the order and labelling that matters. For this reason, we tacitly assume that whenever we have two pomsets, we pick representatives that have disjoint carrier sets.

Definition 2.2. The width of a pomset \( U \), denoted \( ||U|| \), is the size of the largest antichain in \( U \) with respect to \( \leq_U \), i.e., the maximum \( n \in \mathbb{N} \) such that there exist \( u_1, u_2, \ldots, u_n \in U \) that are not related by \( \leq_U \).

The pomsets we work with in this paper have a finite carrier. As a result, \( ||U|| \) is always defined. For instance, the width of the pomset \( C \) above is 2, because the nodes (ii) and (iii) are an antichain of size 2, and there is no antichain of size 3.

Definition 2.3. Let \( U \) and \( V \) be pomsets. The sequential composition of \( U \) and \( V \), denoted \( U \cdot V \), is the pomset \( (U \cup V, \leq_U \cup \leq_V \cup (U \times V), \lambda_U \cup \lambda_V) \). The parallel composition of \( U \) and \( V \), denoted \( U \parallel V \), is the pomset \( (U \cup V, \leq_U \cup \leq_V, \lambda_U \cup \lambda_V) \). Here, \( \lambda_U \cup \lambda_V \) is the function from \( U \cup V \) to \( \Sigma \) that agrees with \( \lambda_U \) on \( U \), and with \( \lambda_V \) on \( V \).

Note that 1 is the unit for both sequential and parallel composition. Sequential composition forces the events in the left pomset to be ordered before those in the right pomset. An example, describing the pomset \( C \cdot C \), is depicted in Figure 1b.

Figure 1 Hasse diagrams for pomsets and a pomset automaton accepting one.

2.1 Pomsets

Partially-ordered multisets, or pomsets [4] for short, generalise words to a setting where events (elements from \( \Sigma \)) may take place not just sequentially, but also in parallel.

Definition 2.1. A labelled poset is a tuple \( (U, \leq_U, \lambda_U) \) consisting of a carrier set \( U \), a partial order \( \leq_U \) on \( U \) and a labelling function \( \lambda_U : U \rightarrow \Sigma \). A labelled poset isomorphism is a bijection between poset carriers that bijectively preserves the labels and the ordering. A pomset is an isomorphism class of labelled posets; equivalently, it is a labelled poset up-to bijective renaming of elements in \( U \). We write \( 1 \) for the empty pomset, \( \text{Pom}_\Sigma \) for the set of all pomsets and \( \text{Pom}_\Sigma^0 \) for the set of all the non-empty pomsets.

For instance, suppose a recipe for caramel-glazed cookies tells us to prepare cookie
dough, bake cookies in the oven, caramelize sugar, and glaze the finished cookies. Here, prepare precedes bake and caramelize, while glaze succeeds both. A pomset representing this process could be \( (C, \leq_C, \lambda_C) \), where \( C = \{ a, b, c \} \) and \( \leq_C \) is such that \( a \leq_C b \leq_C d \) and \( a \leq_C c \leq_C d \), and \( \lambda_C(a) = \text{prepare}, \lambda_C(b) = \text{bake}, \lambda_C(c) = \text{caramelize}, \lambda_C(d) = \text{glaze} \).

Note that words are just finite pomsets with a total order. We will sometimes use \( a \in \Sigma \) to refer to the pomset with a single point labelled \( a \) (and the obvious order); such a pomset is called primitive. A pomset can be represented as a Hasse diagram, where nodes have labels in \( \Sigma \). For instance, the Hasse diagram for the pomset \( C \) above is drawn in Figure 1a.

To simplify notation, we refer to a pomset by the carrier \( U \) of a labelled poset \( (U, \leq_U, \lambda_U) \) in its isomorphism class. We use the symbols \( U, V, W \) and \( X \) to denote pomsets. Pomsets being isomorphism classes, the content of the carrier of the chosen representative is of very little importance; it is the order and labelling that matters. For this reason, we tacitly assume that whenever we have two pomsets, we pick representatives that have disjoint carrier sets.

Definition 2.2. The width of a pomset \( U \), denoted \( ||U|| \), is the size of the largest antichain in \( U \) with respect to \( \leq_U \), i.e., the maximum \( n \in \mathbb{N} \) such that there exist \( u_1, u_2, \ldots, u_n \in U \) that are not related by \( \leq_U \).

The pomsets we work with in this paper have a finite carrier. As a result, \( ||U|| \) is always defined. For instance, the width of the pomset \( C \) above is 2, because the nodes (ii) and (iii) are an antichain of size 2, and there is no antichain of size 3.

Definition 2.3. Let \( U \) and \( V \) be pomsets. The sequential composition of \( U \) and \( V \), denoted \( U \cdot V \), is the pomset \( (U \cup V, \leq_U \cup \leq_V \cup (U \times V), \lambda_U \cup \lambda_V) \). The parallel composition of \( U \) and \( V \), denoted \( U \parallel V \), is the pomset \( (U \cup V, \leq_U \cup \leq_V, \lambda_U \cup \lambda_V) \). Here, \( \lambda_U \cup \lambda_V \) is the function from \( U \cup V \) to \( \Sigma \) that agrees with \( \lambda_U \) on \( U \), and with \( \lambda_V \) on \( V \).

Note that 1 is the unit for both sequential and parallel composition. Sequential composition forces the events in the left pomset to be ordered before those in the right pomset. An example, describing the pomset \( C \cdot C \), is depicted in Figure 1b.
Definition 2.4. The set of series-parallel pomsets, \( \text{Pom}^{\text{sp}}_\Sigma \), is the smallest set that includes the empty and primitive pomsets and is closed under sequential and parallel composition.

In this paper we will be mostly concerned with series-parallel pomsets. For inductive reasoning about them, it is useful to record the following lemma.

Lemma 2.5. Let \( U \in \text{Pom}^{\text{sp}}_\Sigma \). If \( U \) is non-empty, then exactly one of the following is true: \( U = a \) for some \( a \in \Sigma \), or \( U = V \cdot W \) for non-empty \( V, W \in \text{Pom}^{\text{sp}}_\Sigma \), strictly smaller than \( U \), or \( U = V \parallel W \) for non-empty \( V, W \in \text{Pom}^{\text{sp}}_\Sigma \), strictly smaller than \( U \).

2.2 Pomset languages

If a sequential program can exhibit multiple traces, we can group the words that represent these traces into a set called a language. By analogy, we can group the pomsets that represent the traces that arise from a parallel program into a set, which we refer to as a pomset language. Pomset languages are denoted by the symbols \( U \) and \( V \).

For instance, suppose that the recipe for glazed cookies may have an optional fifth step where chocolate sprinkles are spread over the cookies. In that case, there are two pomsets that describe a trace arising from the recipe, \( C^+ \) and \( C^- \), either with or without the chocolate sprinkles. The pomset language \( C = \{ C^- , C^+ \} \) describes the new recipe.

Definition 2.6. Let \( U \) be a pomset language. \( U \) has bounded width if there is \( n \in \mathbb{N} \) such that for all \( U \in U \) we have \( \|U\| \leq n \). The minimal such \( n \) is the width of \( U \), written \( \|U\| \).

The pomset languages considered in this paper have bounded width, and hence \( \|U\| \) is always defined. For instance, the width of \( U \) is 2, because the width of both \( C^- \) and \( C^+ \) is 2.

The sequential and parallel compositions of pomsets can be lifted to pomset languages. We also define a Kleene closure operator, similar to the one defined on languages of words.

Definition 2.7. Let \( U \) and \( V \) be pomset languages. We define:

\[
U \cdot V = \{ U \cdot V : U \in U, V \in V \} \quad U \parallel V = \{ U \parallel V : U \in U, V \in V \}
\]

\[
U^* = \bigcup_{n \in \mathbb{N}} U^n
\]

Where \( U^0 = \{1\} \), and \( U^{n+1} = U \cdot U^n \) for all \( n \in \mathbb{N} \).

Kleene closure models indefinite repetition. For instance, if our cookie recipe has a final step “repeat until enough cookies have been made”, the pomset language \( C^* \) represents all possible traces of repetitions of the recipe; e.g., \( C^+ \cdot C^+ \cdot C^- \in C^* \) is the trace where first two batches of sprinkled cookies are made, followed by one without sprinkles.

2.3 Series-rational expressions

Just like a rational expression can be used to describe a regular structure of sequential events, a series-rational expression can be used to describe a regular structure of possibly parallel events. Series-rational expressions are rational expressions with parallel composition.

Definition 2.8. The series-rational expressions, denoted \( T_\Sigma \), are formed by the grammar

\[
e, f ::= 0 \mid 1 \mid a \in \Sigma \mid e + f \mid e \cdot f \mid e \parallel f \mid e^*.
\]

We use the symbols \( d, e, f, g \) and \( h \) to denote series-rational expressions.

The semantics of a series-rational expression is given by a pomset language.
Definition 2.9. The function \([-\cdot]: \mathcal{T}_\Sigma \to 2^{\text{Pom}}\) is defined inductively, as follows:

\[
\begin{align*}
[0] &= \emptyset & [a] &= \{a\} & [1] &= \{1\} & [e^+] &= [e]^+ \\
[e + f] &= [e] \cup [f] & [e \cdot f] &= [e] \cdot [f] & [e \parallel f] &= [e] \parallel [f]
\end{align*}
\]

If \(\mathcal{U} \in 2^{\text{Pom}}\) such that \(\mathcal{U} = [e]\) for some \(e \in \mathcal{T}_\Sigma\), then \(\mathcal{U}\) is a series-rational language.

To illustrate, consider the pomset language \(C^* = \{C^+, C^-\}\), which describes the possible traces arising from indefinitely repeating the cookie recipe, optionally adding chocolate sprinkles at every repetition. We can describe the pomset language \(\{C^-\}\) with the series-rational expression \(c^- = \text{prepare} \cdot (\text{bake} \parallel \text{caramelize}) \cdot \text{glaze}\), and \(\{C^+\}\) by \(c^+ = c^- \cdot \text{sprinkle}\), which yields the series-rational expression \(c = c^- + c^+\) for \(C\). By construction, \([c^+] = C^*\).

### 2.4 Additive congruence

The following congruence on series-rational expressions will be instrumental in analyzing the automaton we introduce in Section 4, and for restricting said automaton to be finite in Section 4.5.

Definition 2.10. We define \(\simeq\) as the smallest congruence on \(\mathcal{T}_\Sigma\) such that:

\[
\begin{align*}
e_1 + 0 &\simeq e_1 & e_1 + e_1 &\simeq e_1 & e_1 + e_2 &\simeq e_2 + e_1 & e_1 + (e_2 + e_3) &\simeq (e_1 + e_2) + e_3 \\
0 \cdot e_1 &\simeq 0 & e_1 \cdot 0 &\simeq 0 & 0 \parallel e &\simeq 0 & e \parallel 0 &\simeq 0
\end{align*}
\]

When \(\{g, h\}, \{g', h'\} \in (\mathcal{T}_\Sigma^2)\) such that \(g \simeq g'\) and \(h \simeq h'\), we write \(\{g, h\} \simeq \{g', h'\}\).

Thus, when we claim that \(e \simeq e'\), we say that \(e\) is equal to \(e'\), modulo associativity, commutativity and idempotence of \(+\), as well as its unit 0, and possibly annihilation of sequential and parallel composition by 0. Moreover, this congruence is sound with respect to the semantics, and it identifies all expressions that have an empty denotational semantics.

Lemma 2.11. Let \(e, f \in \mathcal{T}_\Sigma\). If \(e \simeq f\), then \([e] = [f]\). Also, \(e \simeq 0\) if and only if \([e] = \emptyset\).

There is a simple linear time decision procedure to test whether two expressions are congruent. This justifies our using this relation to build finite automata later on. As a by-product, we get that the emptiness problem for series-rational expressions is linear time decidable.

### 3 Pomset Automata

We are now ready to describe an automaton model that recognises series-rational languages.

Definition 3.1. A pomset automaton (PA) is a tuple \((Q, \delta, \gamma, F)\) where \(Q\) is a set of states, with \(F \subseteq Q\) the accepting states, \(\delta: Q \times \Sigma \to Q\) is a function called the sequential transition function, \(\gamma: Q \times \binom{Q}{2} \to Q\) is a function called the parallel transition function.

Note that we do not fix an initial state. As a result, a PA does not define a single pomset language but rather a mapping from its states to pomset languages. The language of a state is defined in terms of a trace relation that involves the transitions of both \(\delta\) and \(\gamma\). Here, \(\delta\) plays the same role as in classic finite automata: given a state and a symbol, it returns the new state after reading that symbol. The function \(\gamma\) warrants a bit more explanation. Given a state \(q\) and a binary multiset of states \(\{r, s\}\), \(\gamma\) tells us the state that is reached after reading two input streams in parallel starting at states \(r\) and \(s\), and having both “subprocesses” reach an accepting state. The precise meaning is given in Definition 3.2 below.
Brzozowski Goes Concurrent – A Kleene Theorem for Pomset Languages

We also define the relation $\rightarrow_A$ by $q \xrightarrow{U} A q'$ if and only if $q' = q$ and $U = 1$, or $q \xrightarrow{U} A q'$. The language of $A$ at $q \in Q$, denoted $L_A(q)$, is the set $\{ U : \exists q' \in F. q \xrightarrow{U} A q' \}$. We say that $A$ accepts the language $U$ if there exists a $q \in Q$ such that $L_A(q) = U$.

Intuitively, $\gamma$ ensures that when a process forks at state $q$ into subprocesses starting at $r$ and $s$, if each of those reaches an accepting state, then the processes can join at $\gamma(q, [r, s])$.

We purposefully omit the empty pomset $\epsilon$ as a label in $\rightarrow_A$; doing so would open up the possibility of having traces of the form $q \xrightarrow{1} A q'$ with $q \neq q'$ (i.e., “silent transitions” or “$\epsilon$-transitions”) for example by defining $\gamma(q, [r, s]) = q'$ for some $r, s \in F$. Avoiding transitions of this kind allows us to prove claims about $\rightarrow_A$ by induction on the pomset size, and leverage Lemma 2.5 in the process to disambiguate between the rules that apply. By extension, we can prove claims about $\rightarrow_A$ and $L_A$ by treating $U = 1$ as a special case.

For the remainder of this section, we fix a PA $A = \langle Q, \delta, \gamma, F \rangle$, and a state $q \in Q$. To simplify matters later on, we assume that $A$ has a state $\perp \in Q - F$ such that, for every $a \in \Sigma$, it holds that $\delta(\perp, a) = \perp$ and, for every $\phi \in \{q_0, prepare\}$, it holds that $\gamma(\perp, \phi) = \perp$. Such a sink state is particularly useful when defining $\gamma$: for a fixed $q \in Q$ not all $[r, s] \in \{q_0\}$ may give a value of $\gamma(q, [r, s])$ that contributes to the language accepted by $q$. In such cases, we can define $\gamma(q, [r, s]) = \perp$. Alternatively, we could have allowed $\gamma$ to be a partial function; we chose $\gamma$ as a total function so as not to clutter the definition of derivatives in Section 4.

We draw a PA in a way similar to finite automata: each state (except $\perp$) is a vertex, and accepting states are marked by a double border. To represent sequential transitions, we draw labelled edges; for instance, in Figure 1c, $\delta(q_0, prepare) = q_1$. To represent parallel transitions, we draw hyper-edges; for instance, in Figure 1c, $\gamma(q_1, \{q_1, q_4\}) = q_2$. To avoid clutter, we do not draw either of these edges types the target state is $\perp$. It is not hard to verify that the pomset $C$ of the earlier example is accepted by the PA in Figure 1c.

In principle, the state space of a PA can be infinite; we use this in Section 4 to define a PA that has all possible series-rational expressions as states. It is however also useful to know when we can prune an infinite PA into a finite PA while preserving the languages of the retained states. In Section 5, we use this to translate the PA to a series-rational expression.

Note that it is not sufficient to talk about reachable states, i.e., states that appear in the target of some trace; we must also include states that are “meaningful” starting points for subprocesses. To do this, we first need a handle on these starting points. Specifically, we are interested in the states where the eventual join of the states yields a state that contributes to the behaviour of the PA, and the states may join again, because they are not the sink state. This is captured in the definition below.

**Definition 3.3.** The support of $q$, written $\pi_A(q)$, is $\{[r, s] \in \{q_0\} : \gamma(q, [r, s]), r, s \neq \perp\}$.

We can now talk about subsets of states of an automaton that are closed, in the sense that the relevant part of a transition function has input and output confined to this set. As a result, we can confine the structure of a given PA to a closed set.

---

1 The relation $\rightarrow_A$ should not be thought of as deterministic; for fixed $q \in Q$ and $U \in \text{Pom}_\Sigma^+$, there may be multiple distinct $q' \in Q$ such that $q \xrightarrow{U} A q'$ – see the extended version [9] for additional information.
Definition 3.4. A set of states $Q' \subseteq Q$ is closed when the following rules are satisfied:

\[
\begin{align*}
\bot & \in Q' & q \in Q' & a \in \Sigma & q \in Q' & \phi \in \pi_A(q) & q \in Q' & \{r,s\} \in \pi_A(q) & r, s \in Q'
\end{align*}
\]

If $Q'$ is closed, the generated sub-PA of $A$ induced by $Q'$, denoted $A \restriction_{Q'}$, is the tuple $(Q', \delta \restriction_{Q'}, \gamma \restriction_{Q'}, Q' \cap F)$ where $\delta \restriction_{Q'}$ and $\gamma \restriction_{Q'}$ are the restrictions of $\delta$ and $\gamma$ to $Q'$.

Because the relevant parts of the transition functions are preserved, it is not surprising that the language of a state in a generated sub-PA coincides with the language of that state in the original PA.

Lemma 3.5. Let $Q' \subseteq Q$ be closed. If $q \in Q'$, then $L_{A\restriction_{Q'}}(q) = L_A(q)$.

We now work out how to find a closed subset of states that contains a particular state. The first step is to characterize the states reachable from $q$ by means of transitions.

Definition 3.6. The reach of $q$, written $\rho_A(q)$, is the smallest set satisfying the rules:

\[
\begin{align*}
q & \in \rho_A(q) & q' & \in \rho_A(q) & a & \in \Sigma & q' & \in \rho_A(q) & \phi & \in \pi_A(q) & q' & \in \rho_A(q) & \gamma(q', \phi) & \in \rho_A(q)
\end{align*}
\]

The reach of a state is closely connected to the states that can be reached from $q$ through the trace relation of the automaton, in the following way:

Lemma 3.7. The set $\rho_A(q) \cup \{\bot\}$ contains $\{q' \in Q : \exists U \in \text{Pom}^+_{\Sigma}. q \xrightarrow{U} A q'\} \cup \{q\}$.

Note that $\rho_A(q) \cup \{\bot\}$ is not necessarily closed: we also need the states required by the fourth rule of closure in Definition 3.4. Thus, if we want to “close” $\rho_A(q) \cup \{\bot\}$ by adding the support of its contents, we need to find closed sets of states that contain branching points. In order to do this inductively, we propose the following subclass of PAs.

Definition 3.8. We say that $A$ is fork-acyclic if there exists a fork hierarchy, which is a strict order $\prec_A \subseteq Q \times Q$ such that the following rules are satisfied:

\[
\begin{align*}
\{r,s\} & \in \pi_A(q) & a & \in \Sigma & r & \prec_A \delta(q, a) & \phi & \in \pi_A(q) & r & \prec_A \gamma(q, \phi)
\end{align*}
\]

The fork hierarchy is connected with the reach of a state in the following way:

Lemma 3.9. Let $q', r \in Q$. If $A$ is fork-acyclic, $q' \in \rho_A(q)$ and $r \prec_A q'$, then $r \prec_A q$.

The term fork-acyclic has been used in literature for similar automata [16, 7]. However, in op. cit., it is defined in terms of the traces that arise from the transition structure of the automaton. In contrast, our definition is purely syntactic: it imposes an order on states such that forks cannot be nested. To show that, as in [16], our definition implies that languages of the PA have bounded width, we present the following lemma. Since the state space of a PA can be infinite, we additionally require that the fork hierarchy is well-founded.

Lemma 3.10. If $A$ is fork-acyclic and $\prec_A$ is well-founded then $L_A(q)$ is of finite width.

We introduce the notion of a bounded PA, which is sufficient to guarantee the existence of a closed, finite subset containing a given state, even when the PA has infinitely many states.

Definition 3.11. Let $A$ be fork-acyclic. We say that $A$ is bounded if $\prec_A$ is well-founded, and for all $q \in Q$, both $\pi_A(q)$ and $\rho_A(q)$ are finite.
Thm 3.12. If A is bounded, then for every state q of A there exists a finite set of states $Q_q \subseteq Q$ that is closed and contains q.

Proof. The proof proceeds by $\prec_A$-induction; this is sound, because $\prec_A$ is well-founded.

Suppose the claim holds for all $r \in Q$ with $r \prec_A q$. If $q' \in \rho_A(q)$ and $\{r, s\} \in \pi_A(q')$, then $r \prec_A q'$ and thus $r \prec_A q$ by Lemma 3.9; by induction we obtain for every such $r$ a finite set of states $Q_r \subseteq Q$ that is closed and contains $r$. We choose:

$$Q_q = \{\bot\} \cup \rho_A(q) \cup \bigcup \{Q_r : q' \in \rho_A(q), \{r, s\} \in \pi_A(q')\}.$$ 

This set is finite because $\rho_A(q)$ and $\pi_A(q')$ are finite for all $q, q' \in Q$ since A is bounded. To see that $Q_q$ is closed, it suffices to show that the last rule of closure holds for $q' \in \rho_A(q)$; it does, since if $q' \in \rho_A(q)$ and $\{r, s\} \in \pi_A(q')$, then $r \in Q_r$ and $s \in Q_s$, thus $r, s \in Q_q$.

4 Expressions to Automata

We now turn our attention to the task of translating a series-rational expression $e$ into a PA that accepts $[e]$. We employ Brzozowski’s method [3] to construct a single syntactic PA where every series-rational expression is a state accepting exactly its denotational semantics. To this end we must define which expressions are accepting, and how the sequential and parallel transition functions transform states – what are, in Brzozowski’s vocabulary, their sequential and parallel derivatives?

We start with the accepting states. In Brzozowski’s construction, a rational expression is accepting if its denotational semantics includes the empty pomset. Analogously, a series-rational expression is accepting if its denotational semantics includes the empty pomset.

Def 4.1. We define the set $F_\Sigma$ to be the smallest subset of $T_\Sigma$ satisfying the rules:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1 \in F_\Sigma$</td>
<td>$e \in F_\Sigma$</td>
</tr>
</tbody>
</table>

It is not hard to see that $e \in F_\Sigma$ if and only if $1 \in [e]$. We use $e \cdot f$ as a shorthand for $f$ if $e \in F_\Sigma$, and 0 otherwise. For an equation $E$, we write $[E]$ as a shorthand for 1 if $E$ holds, and 0 otherwise. We now define sequential and parallel derivatives:

Def 4.2. We define the function $\delta_\Sigma : T_\Sigma \times \Sigma \rightarrow T_\Sigma$ as follows:

$$\delta_\Sigma(0, a) = 0 \quad \delta_\Sigma(1, a) = 0 \quad \delta_\Sigma(b, a) = [a = b] \quad \delta_\Sigma(e^*, a) = \delta_\Sigma(e, a) \cdot e^*$$

$$\delta_\Sigma(e + f, a) = \delta_\Sigma(e, a) + \delta_\Sigma(f, a) \quad \delta_\Sigma(e \cdot f, a) = \delta_\Sigma(e, a) \cdot f + e \cdot \delta_\Sigma(f, a) \quad \delta_\Sigma(e \parallel f, a) = e \cdot \delta_\Sigma(f, a) + f \cdot \delta_\Sigma(e, a)$$

Furthermore, the function $\gamma_\Sigma : T_\Sigma \times \left(\frac{T_\Sigma}{2}\right) \rightarrow T_\Sigma$ is defined as follows:

$$\gamma_\Sigma(0, \phi) = 0 \quad \gamma_\Sigma(1, \phi) = 0 \quad \gamma_\Sigma(b, \phi) = 0 \quad \gamma_\Sigma(e^*, \phi) = \gamma_\Sigma(e, \phi) \cdot e^*$$

$$\gamma_\Sigma(e + f, \phi) = \gamma_\Sigma(e, \phi) + \gamma_\Sigma(f, \phi) \quad \gamma_\Sigma(e \cdot f, \phi) = \gamma_\Sigma(e, \phi) \cdot f + e \cdot \gamma_\Sigma(f, \phi) \quad \gamma_\Sigma(e \parallel f, \phi) = [\phi \simeq [e, f]] + e \cdot \gamma_\Sigma(f, \phi) + f \cdot \gamma_\Sigma(e, \phi)$$

The definition of $\delta_\Sigma$ coincides with Brzozowski’s derivative on rational expressions. The definition of $\gamma_\Sigma$ mimics the definition of $\delta_\Sigma$ on non-parallel terms except $b \in \Sigma$. 
The definition of $\gamma_\Sigma$ on parallel terms includes (in the first term) the possibility that the starting states provided to the parallel transition function are (congruent to) the operands of the parallel, in which case the target join state is the accepting state 1. The other two terms (as well as the definition of $\delta_\Sigma$ on a parallel term) account for the fact that if $1 \in [e]$, then $[f] \subseteq [e \parallel f]$. Since we do not allow traces labelled with the empty pomset, traces that originate from these operands are thus lifted to the composition when necessary.

Definition 4.3. The syntactic PA is the PA $A_\Sigma = \langle T_\Sigma, \delta_\Sigma, \gamma_\Sigma, F_\Sigma \rangle$.

We use $L_\Sigma$ as a shorthand for $L_{A_\Sigma}$, and $\to_\Sigma$ (denoted by $\to_{A_\Sigma}$) as a shorthand for $\to_{A_\Sigma}$.

The remainder of this section is devoted to showing that if $e \in T_\Sigma$, then $L_\Sigma(e) = [e]$.

4.1 Traces of congruent states

In the analysis of the syntactic trace relation $\to_\Sigma$, we often encounter sums of terms. To work with these, it is useful to identify terms modulo $\simeq$. In this section, we establish that such an identification is in fact sound, in the sense that if two expressions are related by $\simeq$, then the languages accepted by the states representing those expressions are also identical.

In the first step towards this goal, we show that $F_\Sigma$ is well-defined with respect to $\simeq$.

Lemma 4.4. Let $e, f \in T_\Sigma$ be such that $e \simeq f$. Then $e \in F_\Sigma$ if and only if $f \in F_\Sigma$.

Also, $\delta_\Sigma$ and $\gamma_\Sigma$ are well-defined with respect to $\simeq$, in the following sense:

Lemma 4.5. Let $e, f \in T_\Sigma$ such that $e \simeq f$. If $a \in \Sigma$, then $\delta_\Sigma(e, a) \simeq \delta_\Sigma(f, a)$. Moreover, if $\phi = \langle g, h \rangle \in \langle T_\Sigma \rangle$ with $g, h \neq 0$, then $\gamma_\Sigma(e, \phi) \simeq \gamma_\Sigma(f, \phi)$, and if $\psi \in \langle T_\Sigma \rangle$ with $\phi \simeq \psi$, then $\gamma_\Sigma(e, \phi) \simeq \gamma_\Sigma(e, \psi)$.

With these lemmas in hand, we can show that $\simeq$ is a “bisimulation” with respect to $\to_\Sigma$.

Lemma 4.6. Let $e, f \in T_\Sigma$ be such that $e \simeq f$. If $e \equiv_\Sigma e'$, then there exists an $f' \in T_\Sigma$ such that $f \equiv_\Sigma f'$ and $e' \simeq f'$.

Let $I$ be a finite set, and let $(e_i)_{i \in I}$ be an $I$-indexed family of terms. In the sequel, we treat $\sum_{i \in I} e_i$ as a term, where the $e_i$ are summed in some arbitrary order or bracketing. The lemmas above guarantee that the precise choice of representing this sum as a term makes no matter with regard to the traces allowed.

4.2 Trace deconstruction

We proceed with a series of lemmas that characterise reachable states in the syntactic PA. More precisely, we show that the expressions reachable from some expression $e$ can be written as sums of expressions reachable from subexpressions of $e$. For this reason, we refer to these observations as trace deconstruction lemmas: they deconstruct a trace of an expression into traces of “smaller” expressions. The purpose of these lemmas is twofold; in Section 4.4, they are used to characterise the languages of expressions as they appear in the syntactic PA, while in Section 4.5 they allow us to bound the reach of an expression.

We start by analysing the traces that originate in base terms, such as 0, 1, or $a \in \Sigma$.

Lemma 4.7. Let $e, e' \in T_\Sigma$ and $U \in \text{Pom}_\Sigma$ such that $e \equiv_{\Sigma} e'$. If $e \in \{0, 1\}$, then $e' = 0$. Furthermore, if $e = b \in \Sigma$, then either $e' = 1$ and $U = b$, or $e' = 0$. 
Note, however, that 0 and 1 are not indistinguishable, for \(0 \not\in F_2\) while \(1 \in F_2\).

We also consider the traces that originate in a sum of terms. The intuition here is that the input is processed by both terms simultaneously, and thus the target state must be the sum of the states that are the result of processing the input for each term individually.

- **Lemma 4.8.** Let \(e_1, e_2 \in T_{\Sigma}^+\) and \(U \in \text{Pom}_{\Sigma}^\circ\). If \(e_1 + e_2 \xrightarrow[\Sigma]{} e'\), then there exist \(e_1', e_2' \in T_{\Sigma}^+\) such that \(e' = e_1' + e_2'\), and \(e_1 \xrightarrow[\Sigma]{} e_1'\) and \(e_2 \xrightarrow[\Sigma]{} e_2'\).

We now consider the traces starting in a sequential composition. The intuition here is that the syntactic PA must first proceed through the left operand, before it can proceed to process the right operand. Thus, either the pomset is processed by the left operand entirely, or we should be able to split the pomset in two sequential parts: the first part is processed by the left operand, and the second by the right operand.

- **Lemma 4.9.** Let \(e_1, e_2 \in T_{\Sigma}^+\) and \(U \in \text{Pom}_{\Sigma}^\circ\) be such that \(e_1 \cdot e_2 \xrightarrow[\Sigma]{} f\). There exist an \(f' \in T_{\Sigma}^+\) and a finite set \(I\), as well as \(I\)-indexed families \((f'_i)_{i \in I}\) over \(F_\Sigma\) and \((f_i)_{i \in I}\) over \(T_{\Sigma}^+\), and \(I\)-indexed families \((U'_i)_{i \in I}\), \((U_i)_{i \in I}\) over \(\text{Pom}_{\Sigma}^\circ\), such that:
  - \(f \simeq f' \cdot e_1 + \sum_{i \in I} f_i\), and \(e_1 \xrightarrow[\Sigma]{} f'_i\), and
  - for all \(i \in I\), \(e_1 \xrightarrow[f'_i \Sigma]{} f'_i\), \(e_2 \xrightarrow[f_i \Sigma]{} f_i\), and \(U = U'_i \cdot U_i\).

The next deconstruction lemma concerns traces originating in a parallel composition. Intuitively, the syntactic PA either processes parallel components of the pomset, or processes according to one operand, provided that the other operand allows immediate acceptance.

- **Lemma 4.10.** If \(e_1 \parallel e_2 \xrightarrow[\Sigma]{} f\), then there exist \(f_1, f_2, f_3 \in T_{\Sigma}^+\), such that:
  - \(f \simeq f_1 + f_2 + f_3\),
  - either \(f_1 = 0\), or \(e_2 \in F_\Sigma\) and \(e_1 \xrightarrow[\Sigma]{} f_1\),
  - either \(f_2 = 0\), or \(e_1 \in F_\Sigma\) and \(e_2 \xrightarrow[\Sigma]{} f_2\), and
  - either \(f_3 = 0\), or \(f_3 = 1\) and there exist \(f'_1, f'_2 \in F_\Sigma\) and \(U_1, U_2 \in \text{Pom}_{\Sigma}^\circ\) such that \(U = U_1 \parallel U_2\) and \(e_1 \xrightarrow[f'_1 \Sigma]{} f'_1\) and \(e_2 \xrightarrow[f'_2 \Sigma]{} f'_2\).

Finally, we analyse the reachable states of an expression of the form \(e^*\). The intuition here is that, starting in \(e^*\), the PA can iterate traces originating in \(e\) indefinitely. The trace should thus be sequentially decomposable, with each component the label of a trace originating in \(e\). Furthermore, all but the last target state of these traces should be accepting.

- **Lemma 4.11.** If \(e^* \xrightarrow[\Sigma]{} f\), then there exists a finite set \(I\) and an \(I\)-indexed family of finite sets \((J_i)_{i \in I}\), as well as \(I\)-indexed families \((f_i)_{i \in I}\) over \(T_{\Sigma}^+\) and \((U_i)_{i \in I}\) over \(\text{Pom}_{\Sigma}^\circ\), and for all \(i \in I\) also \(J_i\)-indexed families \((f_{i,j})_{j \in J_i}\) over \(F_\Sigma\) and \((U_{i,j})_{j \in J_i}\) over \(\text{Pom}_{\Sigma}^\circ\), such that \(f \simeq \sum_{i \in I} f_i \cdot e^*\), and for all \(i \in I\):
  - \(e \xrightarrow[\Sigma]{} f_i\),
  - for all \(j \in J_i\) we have that \(e \xrightarrow[f_{i,j} \Sigma]{} f_{i,j}\), and
  - \(U = U'_i \cdot U_i\), where \(U'_i\) is some concatenation of all \(U_{i,j}\) for all \(j \in J_i\).

### 4.3 Trace construction

In the above, we learned how to deconstruct traces in the syntactic PA. To verify that the state in the syntactic PA associated with a series-rational expression \(e\) indeed accepts the series-rational pomset language \([e]\), we also need to show the converse, that is, how to **construct** traces in the syntactic PA from smaller traces. In this context it is often useful to work with the preorder obtained from \(\simeq\).
Definition 4.12. The relation \( \preceq \subseteq \mathcal{T}_\Sigma \times \mathcal{T}_\Sigma \) is defined by \( e \preceq f \) if and only if \( e + f \simeq f \).

The intuition to \( e \preceq f \) is that \( e \) consists of one or more terms that also appear in \( f \), up to \( \simeq \).

In analogy to Lemma 4.6, we show that \( \preceq \) is a “simulation” with respect to traces.

Lemma 4.13. Let \( e, e', f \in \mathcal{T}_\Sigma \) be such that \( e \preceq f \). If \( f \mathrel{\rightharpoonup}_\Sigma e' \), then there exists an \( f' \in \mathcal{T}_\Sigma \) such that \( f' \mathrel{\rightharpoonup}_\Sigma e' \) and \( e' \preceq f' \). Furthermore, if \( e \in F_\Sigma \), then \( f \in F_\Sigma \).

The following lemma tells us that we can create a trace labelled with the concatenation of the labels of two smaller traces, and starting in the sequential composition of the original starting states, provided that the first trace ends in an accepting state. Furthermore, the target state of the newly constructed trace contains the target state of the second trace.

Lemma 4.14. Let \( e_1, e_2, f_2 \in \mathcal{T}_\Sigma \) and \( f_1 \in F_\Sigma \). If \( U, V \in \text{Pom}_\Sigma^\mathcal{T} \) are such that \( e_1 \mathrel{\rightharpoonup}_\Sigma f_1 \) and \( e_2 \mathrel{\rightharpoonup}_\Sigma f_2 \), then there exists an \( f \in \mathcal{T}_\Sigma \) such that \( e_1 \cdot e_2 \mathrel{\rightharpoonup}_\Sigma f \) with \( f_2 \subseteq f \).

We can also construct traces that start in a parallel composition. One way is to construct traces that start in each operand and reach an accepting state; we obtain a trace in their parallel composition almost trivially. If one of the operands is accepting, we can also construct a single trace that starts in the other operand and obtain a trace with the same label starting in the parallel construction. In both cases, we describe the target of the new trace using \( \preceq \).

Lemma 4.15. Let \( e_1, e_2 \in \mathcal{T}_\Sigma \). The following hold:
- If \( f_1, f_2 \in F_\Sigma \) and \( U, V \in \text{Pom}_\Sigma^\mathcal{T} \) are such that \( e_1 \mathrel{\rightharpoonup}_\Sigma f_1 \) and \( e_2 \mathrel{\rightharpoonup}_\Sigma f_2 \), then there exists an \( f \in \mathcal{T}_\Sigma \) such that \( e_1 \parallel e_2 \mathrel{\rightharpoonup}_\Sigma f \) with \( 1 \preceq f \).
- If \( e_2 \in F_\Sigma \) (respectively \( e_1 \in F_\Sigma \)), and \( f' \in \mathcal{T}_\Sigma \) and \( U \in \text{Pom}_\Sigma^\mathcal{T} \) are such that \( e_1 \mathrel{\rightharpoonup}_\Sigma f' \) (respectively \( e_2 \mathrel{\rightharpoonup}_\Sigma f' \)), then there exists an \( f \in \mathcal{T}_\Sigma \) such that \( e_1 \parallel e_2 \mathrel{\rightharpoonup}_\Sigma f \) with \( f' \preceq f \).

Lastly, we present a trace construction lemma to obtain traces originating in expressions of the form \( e^* \). The idea here is that, given a finite number of traces that originate in \( e \), where all (but possibly one) have an accepting state as their target, we can construct a trace originating in \( e^* \), with a concatenation of the labels of the input traces as its label.

Lemma 4.16. Let \( e, f_1, f_2, \ldots, f_n \in \mathcal{T}_\Sigma \) (with \( n > 0 \)) be such that \( f_1, f_2, \ldots, f_{n-1} \in F_\Sigma \). Also, let \( U, U_1, U_2, \ldots, U_n \in \text{Pom}_\Sigma^\mathcal{T} \) be such that \( U = U_1 \cdot U_2 \cdots U_n \). If for all \( i \leq n \) it holds that \( e \mathrel{\rightharpoonup}_\Sigma f_i \), then there exists an \( f \in \mathcal{T}_\Sigma \) such that \( e^* \mathrel{\rightharpoonup}_\Sigma f \), with \( f_n \cdot e^* \preceq f \).

4.4 Soundness for the syntactic PA

With trace deconstruction and construction lemmas in our toolbox, we are ready to show that the syntactic PA indeed captures series-rational languages.

First, note that \( L_\Sigma \) can be seen as a function from \( \mathcal{T}_\Sigma \) to \( \text{Pom}_\Sigma^\mathcal{T} \), like \([\cdot] \). To establish equality between \( L_\Sigma \) and \([\cdot] \), we first show that \( L_\Sigma \) enjoys the same homomorphic equalities as those in the definition of the semantic map, i.e., that \( L_\Sigma(e) \) can be expressed in terms of \( L_\Sigma \) applied to subexpressions of \( e \). The proofs of the equalities below follow a similar pattern: for the inclusion from left to right we use trace deconstruction lemmas to obtain traces for the component expressions, while for the inclusion from right to left we use trace construction lemmas to build traces for the composed expressions given the traces of the component expressions. We treat the case for the empty pomset separately almost everywhere.

Lemma 4.17. Let \( e_1, e_2 \in \mathcal{T}_\Sigma \), and \( a \in \Sigma \). The following equalities hold:
- \( L_\Sigma(0) = \emptyset \) \( L_\Sigma(1) = \{ 1 \} \) \( L_\Sigma(a) = \{ a \} \) \( L_\Sigma(e_1 + e_2) = L_\Sigma(e_1) \cup L_\Sigma(e_2) \)
- \( L_\Sigma(e_1 \cdot e_2) = L_\Sigma(e_1) \cdot L_\Sigma(e_2) \) \( L_\Sigma(e_1 \parallel e_2) = L_\Sigma(e_1) \parallel L_\Sigma(e_2) \) \( L_\Sigma(e_1^*) = L_\Sigma(e_1)^* \)
Brzozowski Goes Concurrent – A Kleene Theorem for Pomset Languages

It is now easy to establish that the Brzozowski construction for the syntactic PA is sound with respect to the denotational semantics of series-rational expressions.

Proposition 4.18. Let \( e \in T_{\Sigma} \). Then \( L_{\Sigma}(e) = [e] \).

Proof. The proof proceeds by induction on \( e \). In the base, \( e = 0, e = 1 \) or \( e = a \) for some \( a \in \Sigma \). In all cases, \( L_{\Sigma}(e) = [e] \) by Lemma 4.17. For the inductive step, there are four cases to consider: either \( e = e_1 + e_2, e = e_1 \cdot e_2, e = e_1 \parallel e_2 \) or \( e = e_1^* \). In all cases, the claim follows from the induction hypothesis and the definition of \([\cdot]\), combined with Lemma 4.17.

4.5 Bounding the syntactic PA

Ideally, we would like to obtain a single PA with finitely many states that recognizes \([e]\) for a given \( e \in T_{\Sigma} \). Unfortunately, the syntactic PA is not bounded, and thus Theorem 3.12 does not apply. For instance, the requirement that \( p_{\Sigma}(e) \) be finite for \( e \in T_{\Sigma} \) fails; consider the family of distinct terms \( \{e_n\}_{n \in \mathbb{N}} \) defined by \( e_0 = 1 \cdot a^* \) and \( e_{n+1} = 0 \cdot a^* + e_n \) for \( n \in \mathbb{N} \); it is not hard to show that \( e_n \in p_{\Sigma}(a^*) \) for \( n \in \mathbb{N} \), and thus conclude that \( p_{\Sigma}(a^*) \) is infinite. We remedy this problem by quotienting the state space of the syntactic PA by congruence.

In what follows, we write \([e]\) for the congruence class of \( e \in T_{\Sigma} \) modulo \( \simeq \), i.e., the set of all \( e' \in T_{\Sigma} \) such that \( e \simeq e' \). We furthermore write \( Q_{\Sigma} \) for the set of all congruence classes of expressions in \( T_{\Sigma} \). We now leverage Lemma 4.5 to define a transition structure on \( Q_{\Sigma} \).

To save space, this section only summarizes the main stepping stones towards finding a finite PA for an expression; for a full proof, we refer to the full version of this paper [9].

Definition 4.19. We define \( \delta_{\simeq} : Q_{\Sigma} \times \Sigma \to Q_{\Sigma} \) and \( \gamma_{\simeq} : Q_{\Sigma} \times (Q_{\Sigma})^2 \to Q_{\Sigma} \) as

\[
\delta_{\simeq}([e], a) = [\delta_{\Sigma}(e, a)] \quad \gamma_{\simeq}([e], [f], [g]) = \begin{cases} [0] & f \simeq 0 \text{ or } g \simeq 0 \\ [\gamma_{\Sigma}(e, f, g)] & \text{otherwise} \end{cases}
\]

Furthermore, the set \( F_{\simeq} \) is defined to be \( \{[e] : e \in F_\Sigma\} \). The quotiented syntactic PA is the PA \( A_{\simeq} = (Q_{\Sigma}, \delta_{\simeq}, \gamma_{\simeq}, F_{\simeq}) \).

Note that, by virtue of Lemma 4.5 and Lemma 4.4, we have that \( \delta_{\simeq} \) and \( \gamma_{\simeq} \), as well as \( F_{\simeq} \), are well-defined. As before, we abbreviate subscripts, for example by writing \( \rightarrow_{\simeq} \) rather than \( \rightarrow_{A_{\simeq}} \), and \( L_{\simeq} \) rather than \( L_{A_{\simeq}} \). Of course, we also want the quotiented syntactic PA to accept the same languages as the syntactic PA. This turns out to be the case.

Theorem 4.20. Let \( e \in T_{\Sigma} \). Then \( L_{\Sigma}(e) = L_{\simeq}([e]) \).

Furthermore, the quotiented syntactic PA is sufficiently restricted to show the following:

Theorem 4.21. The quotiented syntactic PA is fork-acyclic and bounded.

The desired result then follows from the above, Lemma 3.5 and Theorem 3.12.

Corollary 4.22. Let \( e \in T_{\Sigma} \). There exists a finite PA \( A_e \) that accepts \([e]\).

5 Automata to expressions

To associate with every state \( q \) in a bounded PA \( A = (Q, \delta, \gamma, F) \) a series-rational expression \( e_q \) such that \([e_q] = L_A(q)\), we modify the procedure for associating a rational expression with a state in a finite automaton described in [12]. The modification consists of adding
parallel terms to the expression associated with \( q \) whenever a fork in \( q \) contributes to its language, i.e., whenever \( \{ r, s \} \in \pi_A(q) \).

In view of the special treatment of 1 in the semantics of PAs, it is convenient to first define expressions \( e_q^+ \) with the property that \([e_q^+] = L_A(q) - \{1\}\); then we can define \( e_q \) by \( e_q = e_q^+ + [q \in F] \). The definition of \( e_q^+ \) proceeds by induction on the well-founded partial order \( \prec_A \) associated with a bounded PA. That is, when defining \( e_q^+ \) we assume the existence of expressions \( e_{q'}^+ \) for all \( q' \in Q \) such that \( q' \prec_A q \).

First, however, we shall define auxiliary expressions \( e_{qq'}^Q \) for suitable choices of \( Q' \subseteq Q \) and of \( q, q' \in Q \). Intuitively, \( e_{qq'}^Q \) denotes the pomset language characterizing all paths from \( q \) to \( q' \) with all intermediate states in \( Q' \); \( e_q^+ \) can then be defined as the summation of all \( e_{qq'}^Q \) with \( q' \in F \cap \rho_A(q) \).

**Definition 5.1.** Let \( Q' \) be a finite subset of \( Q \), and assume that for all \( r \in Q \) such that \( r \prec_A q \) for some \( q \in Q' \) there exists a series-rational expression \( e_r^+ \in T_\Sigma \) such that \([e_r^+] = L_A(q) - \{1\}\). For all \( Q' \subseteq Q' \) and \( q, q' \in Q' \), we define a series-rational expression \( e_{qq'}^{Q'} \) by induction on the size of \( Q' \), as follows:

1. If \( Q'' = \emptyset \), then let \( \bar{\Sigma} = \{ a \in \Sigma : q' = \delta(q, a) \} \), and let \( \bar{Q} = \{ \phi \in \pi_A(q) : \gamma(q, \phi) = q' \} \). We define
   \[
   e_{qq'}^{Q''} = \sum_{a \in \bar{\Sigma}} a + \sum_{q, r \in \bar{Q}} e_r^+ \parallel e_s^+ .
   \]

2. Otherwise, we choose a \( q'' \in Q'' \) and define
   \[
   e_{qq'}^{Q''} = e_{qq''}(q'') + e_{qq''}(q'') \cdot (e_{qq''}(q'')^* \cdot e_{qq''}(q'')^* .
   \]

Note that \( e_r^+ \) and \( e_s^+ \), appearing in the first clause of the definition of \( e_{qq'}^{Q''} \), exist by assumption, for by fork-acyclicity we have that \( r, s \prec_A q \in Q' \).

**Theorem 5.2.** Let \( Q' \) be a finite subset of \( Q \) and assume that for all \( r \in Q \) such that \( r \prec_A q \) for some \( q \in Q' \) there exists a series-rational expression \( e_r^+ \in T_\Sigma \) such that \([e_r^+] = L_A(q) - \{1\}\). For all \( q, q' \in Q' \), for all \( Q'' \subseteq Q' \), and for all \( U \in Pomo^+_\Sigma \), we have that \( q \mid_{\Sigma} q' \) according to some path that only visits states in \( Q'' \) if, and only if, \( U \in e_{qq'}^{Q''} \).

Using the auxiliary expressions \( e_{qq'}^{Q''} \), we can now associate series-rational expressions \( e_q, e_q^+ \in T_\Sigma \) with every \( q \in Q \), defining \( e_q^+ \) by \( e_q^+ = \sum_{q' \in \rho_A(q) \cap F} e_{qq'}^{\rho_A(q)} \) and \( e_q = e_q^+ + [q \in F] \). Note that \( q \in \rho_A(q) \) and, by Lemma 3.9, for all \( q' \in Q \) such that \( q' \prec_A q'' \) for some \( q'' \in \rho_A(q) \) we have \( q' \prec_A q \), and hence there exists, by induction, a series-rational expression \( e_{q'}^+ \in T_\Sigma \) such that \([e_{q'}^+] = L_A(q') \). So the expressions \( e_{qq'}^{\rho_A(q)} \) are, indeed, defined in Definition 5.1.

**Corollary 5.3.** For every state \( q \in Q \) we have \([e_q^+] = L_A(q) - \{1\}\) and \([e_q] = L_A(q)\).

## 6 Discussion

Another automaton formalism for pomsets, *branching automata*, was proposed by Lodaya and Weil \[15, 16\]. Branching automata define the states where parallelism can start (fork) or end (join) in two relations; pomset automata condense this information in a single function. Lodaya and Weil also provided a translation of series-parallel expressions to branching automata, based on Thompson’s construction \[19\], which relies on the fact that
their automata encode transitions non-deterministically, i.e., as relations. Our Brzozowski-style [3] translation, in contrast, directly constructs transition functions from the expressions. Lastly, their translation of branching automata to series-parallel expressions is only sound for a semantically restricted class of automata, whereas our restriction is syntactic.

Jipsen and Moshier [8] provided an alternative formulation of the automata proposed by Lodaya and Weil, also called branching automata. Their method to encode parallelism in these branching automata is conceptually dual to pomset automata: branching automata distinguish based on the target states of traces to determine the join state, whereas pomset automata distinguish based on the origin states of traces. The translations of series-parallel expressions to branching automata and vice versa suffer from the same shortcomings as those by Lodaya and Weil, i.e., transition relations rather than functions and a semantic restriction on automata for the translation of automata to expressions.

Lodaya and Weil observed [16] that the behaviour of their automata corresponds to 1-safe Petri nets. Since the behavior of their branching automata can be matched with our (bounded, fork-acyclic) pomset automata, we believe that 1-safe Petri nets also correspond to our automata. We opted to treat semantics of series-rational expressions in terms of automata instead of Petri nets to find more opportunities to extend to a coalgebraic treatment. While the present paper does not reach this goal, we believe that our formulation in terms of states and transition functions offers some hope of getting there.

Prisacariu introduced Synchronous Kleene Algebra (SKA) [17], extending Kleene Algebra with a synchronous composition operator. SKA differs from our model in that it assumes that all basic actions are performed in unit time, and that actors responsible for individual actions never idle. In contrast, our (weak BKA-like) model makes no synchrony assumptions; expressions can be composed in parallel, and the relative timing of basic actions within those expressions is irrelevant for the semantics. Prisacariu axiomatized SKA and extended it to Synchronous Kleene Algebra with Tests (SKAT); others [2] proposed Brzozowski-style derivatives of SKA expressions and used them to test equivalence of SKA(T) expressions.

Further work

We plan to extend our results to semantics of series-parallel expressions in terms of downward-closed pomset languages, i.e., sets of pomsets that are closed under Gischer’s subsumption order [4]. Such an extension would correspond to adding the weak exchange law (which relates sequential and parallel compositions), and thus yields an operational model for weak CKA. We conjecture that no change to the automaton model is necessary to accommodate this generalization, just like Struth and Laurence suspect that the downward-closed semantics of series-parallel expressions can be captured by their non-downward closed semantics.

Our series-rational expressions do not include the parallel analogue of the Kleene star (sometimes called “parallel star”, or “replication”). Future work could look into extending derivatives to include this operator, and relaxing fork-acyclicity to allow recovering expressions that include the parallel star from an automaton that satisfies this weaker restriction.

A classic result by Kozen [11] axiomatizes language equivalence of rational expressions using Kleene’s theorem [10] and the uniqueness of minimal finite automata; consequently, the free model for KA can also be characterized in terms of rational languages. It would be interesting to see if the same technique can be used (based on pomset automata) to show that the axioms of weak Bi-Kleene Algebra are a complete axiomatization of pomset language equivalence of series-rational expressions, and thus characterise the free weak Bi-Kleene Algebra (or even the free weak CKA) in terms of series-rational pomset languages. Although such a result was recently published [14], it does not rely on an automaton model.
Brzozowski derivatives for classic rational expressions induce a coalgebra on rational expressions that corresponds to a finite automaton. We aim to study series-rational expressions coalgebraically. The first step would be to find the coalgebraic analogue of pomset automata such that language acceptance is characterized by the homomorphism into the final coalgebra. Ideally, such a view of pomset automata would give rise to a decision procedure for equivalence of series-rational expressions based on coalgebraic bisimulation-up-to [18].

Rational expressions can be extended with tests to reason about imperative programs equationally [13]. In the same vein, one can extend series-rational expressions with tests [7, 8] to reason about parallel imperative programs equationally. We are particularly interested in employing such an extension to extend the network specification language NetKAT [1] with primitives for concurrency so as to model and reason about concurrency within networks.

Acknowledgements. We thank the anonymous reviewers for their insightful comments.

References


Algebraic Laws for Weak Consistency

Andrea Cerone\textsuperscript{1}, Alexey Gotsman\textsuperscript{2}, and Hongseok Yang\textsuperscript{3}

\textsuperscript{1} Imperial College London, UK
\texttt{a.cerone@imperial.ac.uk}
\textsuperscript{2} IMDEA Software Institute, Madrid, Spain
\texttt{alexey.gotsman@imdea.org}
\textsuperscript{3} University of Oxford, UK
\texttt{hongseok.yang@cs.ox.ac.uk}

Abstract
Modern distributed systems often rely on so called weakly consistent databases, which achieve scalability by weakening consistency guarantees of distributed transaction processing. The semantics of such databases have been formalised in two different styles, one based on abstract executions and the other based on dependency graphs. The choice between these styles has been made according to intended applications. The former has been used for specifying and verifying the implementation of the databases, while the latter for proving properties of client programs of the databases. In this paper, we present a set of novel algebraic laws (inequalities) that connect these two styles of specifications. The laws relate binary relations used in a specification based on abstract executions to those used in a specification based on dependency graphs. We then show that this algebraic connection gives rise to so called robustness criteria: conditions which ensure that a client program of a weakly consistent database does not exhibit anomalous behaviours due to weak consistency. These criteria make it easy to reason about these client programs, and may become a basis for dynamic or static program analyses. For a certain class of consistency models specifications, we prove a full abstraction result that connects the two styles of specifications.

1 Introduction
Modern distributed systems often rely on databases that achieve scalability by weakening consistency guarantees of distributed transaction processing. These databases are said to implement weak consistency models. Such weakly consistent databases allow for faster transaction processing, but exhibit anomalous behaviours, which do not arise under a database with a strong consistency guarantee, such as serialisability. Two important problems for the weakly consistent databases are: (i) to find elegant formal specifications of their consistency models and to prove that these specifications are correctly implemented by protocols used in the databases; (ii) to develop effective reasoning techniques for applications running on top of such databases. These problems have been tackled by using two different formalisms, which model the run-time behaviours of weakly consistent databases differently.

When the goal is to verify the correctness of a protocol implementing a weak consistency model, the run-time behaviour of a distributed database is often described in terms of abstract executions \cite{13}, which abstract away low-level implementation details of the database (Section 2). An example of abstract execution is depicted in Figure 1; ignore the bold edges for the moment. It comprises four transactions, $T_0$, $T_1$, $T_2$, and $S$; transaction $T_0$ initializes
the value of an object acct to 0; transactions T₁ and T₂ increment the value of acct by 50 and 25, respectively, after reading its initial value; transaction S reads the value of acct. In this abstract execution, both the updates of T₁ and T₂ are VISible to transaction S, as witnessed by the two VIS-labelled edges: T₁ VIS S and T₂ VIS S.

On the other hand, the update of T₁ is not visible to T₂, and vice versa, as indicated by the absence of an edge labelled with VIS between these transactions. Intuitively, the absence of such an edge means that T₁ and T₂ are executed concurrently. Because S sees T₁ and T₂, as indicated by VIS-labelled edges from T₁ and T₂ to S, the result of reading the value of acct in S must be one of the values written by T₁ and T₂. However, because these transactions are concurrent, there is a race, or conflict, between them. The AR-labelled edge connecting T₁ to T₂, is used to ARbitrate the conflict: it states that the update of T₁ is older than the one of T₂, hence the query of acct in S returns the value written by the latter.

The style of specifications of consistency models in terms of abstract executions can be given by imposing constraints over the relations VIS, AR (Section 2.1). A set of transactions \( T = \{T₁, T₂, \cdots \} \), called a history, is allowed by a consistency model specification if it is possible to exhibit two witness relations VIS, AR over \( T \) such that the resulting abstract execution satisfies the constraints imposed by the specification. For example, serialisability can be specified by requiring that the relation VIS should be a strict total order. The set of transactions \( \{T₀, T₁, T₂, S\} \) from Figure 1 is not serialisable: it is not possible to choose a relation VIS such that the resulting abstract execution relates the transactions T₁, T₂ and the results of read operations are consistent with visible updates.

Specifications of consistency models using abstract executions have been used in the work on proving the correctness of protocols implementing weak consistency models, as well as on justifying operational, implementation-dependent descriptions of these models [11, 12, 13, 15].

The second formalism used to define weak consistency models is based on the notion of dependency graphs [1], and it has been used for proving properties of client programs running on top of a weakly consistent database. Dependency graphs capture the data dependencies of transactions at run-time (Section 3): the transactions \( \{T₀, T₁, T₂, S\} \) depicted above, together with the bold edges but without normal edges, constitute an example of dependency graph. The edge \( T₂ \xrightarrow{\text{WR(acct)}} S \) denotes a write-read dependency. It means that the read of acct in transaction S returns the value written by transaction T₂, and the edges \( T₀ \xrightarrow{\text{WR(acct)}} T₁ \) and \( T₀ \xrightarrow{\text{WR(acct)}} T₂ \) mean something similar. The edge \( T₁ \xrightarrow{\text{WW(acct)}} T₂ \) denotes a write-write dependency, and says that the write to acct in T₂ supersedes the write to the same object in

\[ \text{Figure 1} \text{ An example of abstract execution and of dependency graph.} \]

\[ ^1 \text{ For simplicity, references to the object acct have been removed from the dependencies of Figure 1.} \]
The remaining edges $T_1 \xrightarrow{RW(acct)} T_2$ and $T_2 \xrightarrow{RW(acct)} T_1$ express anti-dependencies. The former means that $T_1$ reads a value for object $acct$ which is older than the value written by $T_2$.

When using dependency graphs, consistency models are specified as sets of transactions for which there exist WR, WW, RW relations that satisfy certain properties, usually stated as particular relations being acyclic [7, 16]; for example, serialisability can be specified by requiring that dependency graphs are acyclic. Because dependencies of transactions can be over-approximated at the compilation time, specifications of consistency models in terms of dependency graphs have been widely used for manually or automatically reasoning about properties of client programs of weakly consistent databases [19, 27]. They have also been used in the complexity and undecidability results for verifying implementations of consistency models [9].

Our ultimate aim is to reveal a deep connection between these two styles of specifying weak consistency models, which was hinted at for specific consistent models in the literature. Such a connection would, for instance, give us a systematic way to derive a specification of a weak consistency model based on dependency graphs from the specification based on abstract executions, while ensuring that the original and the derived specifications are equivalent in a sense. In doing so, it would enable us to prove properties about client programs of a weakly consistent database using techniques based on dependency graphs [9, 16, 18] even when the consistency model of the database is specified in terms of abstract executions.

In this paper, we present our first step towards this ultimate aim. First, we observe that each abstract execution determines an underlying dependency graph. Then we study the connection between these two structures at an algebraic level. We propose a set of algebraic laws, parametric in the specification of a consistency model to which the original abstract execution belongs (Section 4). These laws can be used to derive properties of the form $R_G \subseteq R_A$: here $R_G$ is an expression from the Kleene Algebra with Tests [22] whose ground terms are run-time dependencies of transactions, and tests are properties over transactions. The relation $R_A$ is one of the fundamental relations of abstract executions: VIS, AR, or a novel relation $VIS^{-1}$ that we call anti-visibility, defined as $VIS^{-1} = \{(T, S) | \neg(S \xrightarrow{VIS} T)\}$. Some of the algebraic laws that we propose show that there is a direct connection between each kind of dependencies and the relations of abstract executions: $WR \subseteq VIS, WW \subseteq AR$, and $RW \subseteq VIS^{-1}$. The other laws capture the connection between the relations of abstract executions $VIS, AR$, and $VIS^{-1}$. The exact nature of this connection depends on the specification of the consistency model of the considered abstract execution.

We are particularly interested in deriving properties of the form $R_G \subseteq AR$. Properties of this form give rise to so called robustness criteria for client programs, conditions ensuring that a program only exhibits serialisable behaviours even when it runs under a weak consistency model [7, 10, 19]. Because AR is a total order, this implies that $R_G$ must be acyclic, hence all cycles must be in the complement of $R_G$. We can then check for the absence of such critical cycles at compile time: because dependency graphs of serialisable databases are always acyclic, this ensures that said application only exhibits serialisable behaviours.

As another contribution we show that, for a relevant class of consistency models, our algebraic laws can be used to derive properties which are not only necessary, but also sufficient, for dependency graphs in such models (Section 5).
2 Abstract Executions

We consider a database storing objects in \( \text{Obj} = \{x, y, \cdots\} \), which for simplicity we assume to be integer-valued. Client programs can interact with the database by executing operations from a set \( \text{Op} \), grouped inside transactions. We leave the set \( \text{Op} \) unspecified, apart from requiring that it contains read and write operations over objects: \( \{\text{write}(x, n), \text{read}(x, n) \mid x \in \text{Obj}, n \in \mathbb{N}\} \subseteq \text{Op} \).

**Histories.** To specify a consistency model, we first define the set of all client-database interactions allowed by the model. We start by introducing (run-time) transactions and histories, which record such interactions in a single computation. Transactions are elements from a set \( T = \{T, S, \cdots\} \); the operations executed by transactions are given by a function \( \text{behav} : T \to 2^{\text{Op}} \), which maps a transaction \( T \) to a set of operations that are performed by the transaction and can be observed by other transactions. We often abuse notations and just write \( o \in T \) (or \( T \ni o \)) instead of \( o \in \text{behav}(T) \). We adopt similar conventions for \( \mathcal{O} \subseteq \text{behav}(T) \) and \( \mathcal{O} = \text{behav}(T) \) where \( \mathcal{O} \) is a subset of operations.

We assume that transactions enjoy atomic visibility: for each object \( x \), (i) a transaction \( S \) never observes two different writes to \( x \) from a single transaction \( T \) and (ii) it never reads two different values of \( x \). Formally, the requirements are that if \( T \ni (\text{write} x : m) \) and \( T \ni (\text{read} x : m) \), or \( T \ni (\text{read} x : n) \) and \( T \ni (\text{read} x : m) \), then \( n = m \). Our treatment of atomic visibility is taken from our previous work on transactional consistency models [15]. Atomic visibility is guaranteed by many consistency models [5, 19, 28]. We point out that although we focus on transactions in distributed systems in the paper, our results apply to weak shared-memory models [4]; there a transaction \( T \) is the singleton set of a read operation \( (T = \{\text{read} x : m\}) \), that of a write operation \( (T = \{\text{write} x : n\}) \), or the set of read and write representing a compare and set operation \( (T = \{\text{read} x : n, \text{write} x : m\}) \).

For each object \( x \), we let \( \text{Writes}_x := \{T \mid \exists n. (\text{write} x : n) \in T\} \) and \( \text{Reads}_x := \{T \mid \exists n. (\text{read} x : n) \in T\} \) be the sets of transactions that write to and read from \( x \), respectively.

**Definition 1.** A history \( T \) is a finite set of transactions \( \{T_1, T_2, \cdots, T_n\} \).

**Consistency Models.** A consistency model \( \Gamma \) is a set of histories that may arise when client programs interact with the database. To define \( \Gamma \) formally, we augment histories with two relations, called visibility and arbitration.

**Definition 2.** An abstract execution \( X \) is a tuple \( (T, \text{VIS}, \text{AR}) \) where \( T \) is a history and \( \text{VIS}, \text{AR} \subseteq (T \times T) \) are relations on transactions such that \( \text{VIS} \subseteq \text{AR} \) and \( \text{AR} \) is a strict total order\(^2\).

We often write \( T \xrightarrow{\text{VIS}} S \) for \( (T, S) \in \text{VIS} \), and similarly for other relations. For each abstract execution \( X = (T, \text{VIS}, \text{AR}) \), we let \( T_X := T, \text{VIS}_X := \text{VIS} \), and \( \text{AR}_X := \text{AR} \).

In an abstract execution \( X, T \xrightarrow{\text{VIS}} S \) means that the read operations in \( S \) may depend on the updates of \( T \), while \( T \xrightarrow{\text{AR}} S \) means that the update operations of \( S \) supersede those performed by \( T \). Naturally, one would expect that the value fetched by read operations in a transaction \( T \) is the most up-to-date one among all the values written by transactions visible to \( T \). For simplicity, we assume that such a transaction always exists.

\(^2\) A relation \( R \subseteq T \times T \) is a strict (partial) order if it is transitive and irreflexive; it is total if for any \( T, S \in T \), either \( T = S \) or \( (T, S) \in R \) or \( (S, T) \in R \).
Definition 3. An abstract execution $\mathcal{X} = (T, \text{VIS}, \text{AR})$ respects the Last Write Win (LWW) policy, if for all $T \in \mathcal{T}$ such that $T \ni (\text{read } x : n)$, the set $T' := (\text{VIS}^{-1}(T) \cap \text{Writes}_x)$ is not empty, and $\max_{\text{AR}}(T') \ni (\text{write } x : n)$, where $\max_{\text{AR}}(T')$ is the AR-supremum of $T'$.

Definition 4. An abstract execution $\mathcal{X} = (T, \text{VIS}, \text{AR})$ respects causality if $\text{VIS}$ is transitive. Any abstract execution that respects both causality and the LWW policy is said to be valid.

We always assume an abstract execution to be valid, unless otherwise stated. Causality is respected by all abstract executions allowed by several interesting consistency models. They also simplify the mathematical development of our results. In [17, Appendix B], we explain how our results can be generalised for consistency models that do not respect causality. We also discuss how the model can be generalised to account for sessions and session guarantees [29].

We can specify a consistency model using abstract executions in two steps. First, we identify properties on abstract executions, or axioms, that formally express an informal consistency guarantee, and form a set with the abstract executions satisfying the properties. Next, we project abstract executions in this set to underlying histories, and define a consistency model $\Gamma$ to be the set of resulting histories.

Abstract executions hide low-level operational details of the interaction between client programs and weakly consistent databases. This benefit has been exploited for proving that such databases implement intended consistency models [11, 12, 13, 15, 20].

2.1 Specification of Weak Consistency Models

In this section we introduce a simple framework for specifying consistency models using the style of specification discussed above. In our framework, axioms of consistency models relate the visibility and arbitration relations via inequalities of the form $R_1 ; \text{AR} \in X ; R_2 \subseteq \text{VIS}_X$, where $R_1$ and $R_2$ are particular relations over transactions, and $\mathcal{X}$ is an abstract execution. As we will explain later, axioms of this form establish a necessary condition for two transactions in an abstract execution $\mathcal{X}$ to be related by $\text{VIS}_X$, i.e. they cannot be executed concurrently. Despite its simplicity, the framework is expressive enough to capture several consistency models for distributed databases [15, 23]; as we will show in Section 4, one of the benefits of this simplicity is that we can infer robustness criteria of consistency models in a systematic way.

As we will see, the relations $R_1, R_2$ in axioms of the form above, may depend on the visibility relation of the abstract execution $\mathcal{X}$. To define such relations, we introduce the notion of specification function.

Definition 5. A function $\rho : 2^{(\mathcal{T} \times \mathcal{T})} \rightarrow 2^{(\mathcal{T} \times \mathcal{T})}$ is a specification function if for every history $\mathcal{T}$ and relation $R \subseteq \mathcal{T} \times \mathcal{T}$, then $\rho(R) = \rho(T \times T) \cap R^\circ$. Here $R^\circ$ is the reflexive closure of $R$. A consistency guarantee, or simply guarantee, is a pair of specification functions $(\rho, \pi)$.

Definition 5 ensures that specification functions are defined locally: for any $R_1, R_2 \subseteq \mathcal{T} \times \mathcal{T}$, $\rho(R_1 \cup R_2) = \rho(R_1) \cup \rho(R_2)$, and in particular for any $R \subseteq \mathcal{T} \times \mathcal{T}$, $\rho(R) = \left( \bigcup_{T,S \in \mathcal{T}} \rho([\{(T,S)\}]) \right) \cap R^\circ$. The reflexive closure in Definition 5 is needed because we will always apply specification functions to irreflexive relations (namely, the visibility relation of abstract executions), although the result of this application need not be irreflexive. For example, $\rho_{\text{id}}(R) := \text{id}$, where $\text{id}$ is the identity function, is a valid specification function.

Each consistency guarantee $(\rho, \pi)$ defines, for each abstract execution $\mathcal{X}$, an axiom of the form $\rho(\text{VIS}_X) ; \text{AR}_X ; \pi(\text{VIS}_X) \subseteq \text{VIS}_X$: if this axiom is satisfied by $\mathcal{X}$, we say that $\mathcal{X}$
Guarantee Associated Axiom

<table>
<thead>
<tr>
<th>Function</th>
<th>Definition</th>
<th>Guarantee</th>
<th>Associated Axiom</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \rho_\text{a}(R) )</td>
<td>( \text{Id} )</td>
<td>( (\rho_\text{a}, \rho_\text{a}) )</td>
<td>AR ( \subseteq ) VIS</td>
</tr>
<tr>
<td>( \rho_\text{s}(R) )</td>
<td>( R \setminus \text{Id} )</td>
<td>( (\rho_\text{a}, \rho_\text{s}) )</td>
<td>AR ; VIS ( \subseteq ) VIS</td>
</tr>
<tr>
<td>( \rho_\text{x}(R) )</td>
<td>( [\text{Writes}_x] )</td>
<td>( (\rho_\text{x}, \rho_\text{x}) )</td>
<td>AR ( ; ) [\text{Writes}_x] ( \subseteq ) VIS</td>
</tr>
<tr>
<td>( \rho_\text{s}(R) )</td>
<td>( [\text{SerTx}] )</td>
<td>( (\rho_\text{s}, \rho_\text{s}) )</td>
<td>AR ( ; ) [\text{SerTx}] ( \subseteq ) VIS</td>
</tr>
</tbody>
</table>

**Figure 2** Some Specification Functions and Consistency Guarantees.

satisfies the consistency guarantee \((\rho, \pi)\). Consistency guarantees impose a condition on when two transactions \(T, S\) in an abstract execution \(\mathcal{X}\) are not allowed to execute concurrently, i.e. they must be related by a VIS\(X\) edge. By definition, in abstract executions visibility edges cannot contradict arbitration edges, hence it is only natural that the order in which the transactions \(T, S\) above are executed is determined by the arbitration order: in fact, the definition of specification function ensures that \(\rho(\text{VIS}_X) \subseteq \text{VIS}_X\) and \(\pi(\text{VIS}_X) \subseteq \text{VIS}_X\), so that \((\rho(\text{VIS}_X) ; \text{AR}_X ; \pi(\text{VIS}_X)) \subseteq \text{AR}_X\) for all abstract executions \(\mathcal{X}\).

**Definition 6.** A consistency model specification \(\Sigma\) or \(x\)-specification is a set of consistency guarantees \(\{(\rho_i, \pi_i)\}_{i \in I}\) for some index set \(I\).

We define \(\text{Executions}(\Sigma)\) to be the set of valid abstract executions that satisfy all the consistency guarantees of \(\Sigma\). We let \(\text{modelOf}(\Sigma) := \{\mathcal{T}_X \mid \mathcal{X} \in \text{Executions}(\Sigma)\}\).

**Examples of Consistency Model Specifications**

Figure 2 shows several examples of specification functions and consistency guarantees. In the figure we use the relations \([T] := \{(T, T) \mid T \in T\}\) and \([o] := \{(T, T) \mid T \ni o\}\) for \(T \subseteq \mathbb{T}\) and \(o \in \text{Op}\). The guarantees in the figure can be composed together to specify, among others, several of the consistency models considered in [15]: we give some examples of them below. Each of these consistency models allows different kinds of anomalies: due to lack of space, these are illustrated in [17, Appendix A].

**Causal Consistency [24].** This is the weakest consistency model we consider. It is specified by \(\Sigma_{CC} = \emptyset\). In this case, all abstract executions in \(\text{Executions}(\Sigma_{CC})\) respect causality. The execution in Figure 1 is an example in \(\text{Executions}(\Sigma_{CC})\).

**Red-Blue Consistency [23].** This model extends causal consistency by marking a subset of transactions as serialisable, and ensuring that no two such transactions appear to execute concurrently. We model red-blue consistency via the \(x\)-specification \(\Sigma_{RB} = \{(\rho_{S}, \rho_{S})\}\). In the definition of \(\rho_{S}\), an element \(\text{SerTx} \in \text{Op}\) is used to mark transactions as serialisable, and the specification requires that in every execution \(\mathcal{X} \in \text{Executions}(\Sigma_{RB})\), any two transactions \(T, S \ni \text{SerTx}\) in \(\mathcal{X}\) be compared by \(\text{VIS}_X\). The abstract execution from Figure 1 is included in \(\text{Executions}(\Sigma_{RB})\), but if it were modified so that transactions \(T_1, T_2\) were marked as serialisable, then the result would not belong to \(\text{Executions}(\Sigma_{RB})\).

**Parallel Snapshot Isolation (PSI) [26, 28].** This model strengthens causal consistency by enforcing the Write Conflict Detection property: transactions writing to one same object do not execute concurrently. We let \(\Sigma_{PSI} = \{(\rho_{x}, \rho_{x})\}_{x \in \text{Obj}}\); every execution \(\mathcal{X} \in \text{Executions}(\Sigma_{PSI})\) satisfies the inequality \((\text{[Writes}_x] ; \text{AR}_X ; \text{[Writes}_x]) \subseteq \text{VIS}_X,\) for all \(x \in \text{Obj}\).
Snapshot Isolation (SI) [6]. This consistency model strengthens PSI by requiring that, in executions, the set of transactions visible to any transaction \( T \) is a prefix of the arbitration relation. Formally, we let \( \Sigma_{SI} = \Sigma_{PSI} \cup \{(\rho_{d}, \rho_{S})\} \). The consistency guarantee \((\rho_{d}, \rho_{S})\) ensures that any abstract execution \( \mathcal{X} \in \text{Executions}(SI) \) satisfies the property \((\mathcal{AR}_{\mathcal{X}} ; \text{VIS}_{\mathcal{X}}) \subseteq \text{VIS}_{\mathcal{X}}^{3}\).

Similarly to what we did to specify Red-Blue consistency, we can strengthen SI by allowing the possibility to mark transactions as serialisable. The resulting x-specification is \( \Sigma_{SI+SER} = \Sigma_{SI} \cup \{(\rho_{S}, \rho_{S})\} \). This x-specification captures a fragment of Microsoft SQL server, which allows the user to select the consistency model at which a transaction should run [25].

Serialisability. Executions in this consistency model require the visibility relation to be total. This can be formalised via the x-specification \( \Sigma_{SER} := \{(\rho_{d}, \rho_{d})\} \). Any \( \mathcal{X} \in \text{Executions}(\Sigma_{SER}) \) is such that \( \mathcal{AR}_{\mathcal{X}} \subseteq \text{VIS}_{\mathcal{X}} \), thus enforcing \( \text{VIS}_{\mathcal{X}} \) to be a strict total order.

3 Dependency Graphs

We present another style of specification for consistency models based on dependency graphs, introduced in [1]. These are structures that capture the data-dependencies between transactions accessing one same object. Such dependencies can be over approximated at compilation time. For this reason, they have found use in static analysis [7, 16, 18, 19] for programs running under a weak consistency model.

Definition 7. A dependency graph is a tuple \( \mathcal{G} = (\mathcal{T}, \mathcal{WR}, \mathcal{WW}, \mathcal{RW}) \), where \( \mathcal{T} \) is a history and

1. \( \mathcal{WR} : \text{Obj} \rightarrow 2^{\mathcal{T} \times \mathcal{T}} \) is such that:
   (a) \( \forall T, S \in \mathcal{T}. \forall x. T \xrightarrow{\mathcal{WR}(x)} S \implies T \neq S \land \exists n. (T \exists \text{write} x : n) \land (S \exists \text{read} x : n) \),
   (b) \( \forall S \in \mathcal{T}. \forall x. (S \exists \text{read} x : n) \implies \exists T. T \xrightarrow{\mathcal{WR}(x)} S \),
   (c) \( \forall T, T', S \in \mathcal{T}. \forall x. (T \xrightarrow{\mathcal{WR}(x)} S \land T' \xrightarrow{\mathcal{WR}(x)} S) \implies T = T' \);
2. \( \mathcal{WW} : \text{Obj} \rightarrow 2^{\mathcal{T} \times \mathcal{T}} \) is such that for every \( x \in \text{Obj} \), \( \mathcal{WW}(x) \) is a strict, total order over \( \text{Writes}_{x} \);
3. \( \mathcal{RW} : \text{Obj} \rightarrow 2^{\mathcal{T} \times \mathcal{T}} \) is such that \( S \xrightarrow{\mathcal{RW}(x)} T \) iff \( S \neq T \) and \( \exists T'. T' \xrightarrow{\mathcal{WR}(x)} S \land T' \xrightarrow{\mathcal{WW}(x)} T \).

Given a dependency graph \( \mathcal{G} = (\mathcal{T}, \mathcal{WR}, \mathcal{WW}, \mathcal{RW}) \), we let \( \mathcal{T}_{\mathcal{G}} := \mathcal{T} \), \( \mathcal{WR}_{\mathcal{G}} := \mathcal{WR} \), \( \mathcal{WW}_{\mathcal{G}} := \mathcal{WW} \), \( \mathcal{RW}_{\mathcal{G}} := \mathcal{RW} \). The set of all dependency graphs is denoted as \( \text{Graphs} \).

Sometimes, we commit an abuse of notation and use the symbol \( \mathcal{WR} \) to denote the relation \( \bigcup_{x \in \text{Obj}} \mathcal{WR}(x) \), and similarly for \( \mathcal{WW} \) and \( \mathcal{RW} \). The actual meaning of \( \mathcal{WR} \) will always be clear from the context.

Let \( \mathcal{G} \in \text{Graphs} \). The write-read dependency \( T \xrightarrow{\mathcal{WR}(x)} S \) means that \( S \) reads the value of object \( x \) that has been written by \( T \). By Definition 7, for any transaction \( S \in \text{Read}_{x} \) there exists exactly one transaction \( T \) such that \( T \xrightarrow{\mathcal{WR}(x)} S \). The relation \( \mathcal{WW}(x) \) establishes a total order in which updates over object \( x \) are executed by transactions; its elements are called write-write dependencies. Edges in the relation \( \mathcal{RW}(x) \) take the name of anti-dependencies. \( T \xrightarrow{\mathcal{RW}(x)} S \) means that transaction \( T \) fetches some value for object \( x \), but this is later updated by \( S \). Given an abstract execution \( \mathcal{X} \), we can extract a dependency graph \( \text{graph}(\mathcal{X}) \) such that \( \mathcal{T}_{\text{graph}(\mathcal{X})} = \mathcal{T}_{\mathcal{X}} \).

\(^{3}\) To be precise, the property induced by the guarantee \((\rho_{d}, \rho_{S})\) is \((\mathcal{AR}_{\mathcal{X}} ; (\text{VIS}_{\mathcal{X}} \setminus \text{Id})) \subseteq \mathcal{AR}_{\mathcal{X}} \). However, since \( \text{VIS}_{\mathcal{X}} \) is an irreflexive relation, \( \text{VIS}_{\mathcal{X}} \setminus \text{Id} = \text{VIS}_{\mathcal{X}} \). Also, note that \( \rho(R) = R \) is not a specification function, so we cannot replace the guarantee \((\rho_{d}, \rho_{S})\) with \((\rho_{d}, \rho)\).
Definition 8. Let \( \mathcal{X} = (T, \text{VIS}, \text{AR}) \) be an execution. For \( x \in \text{Obj} \), we define \( \text{graph}(\mathcal{X}) = (T, \text{WR}_X, \text{WW}_X, \text{RW}_X) \), where:

1. \( T \xrightarrow{\text{WR}_X(x)} S \iff (S \ni \text{read } x : \_ ) \land T = \max_{\text{AR}} (\text{VIS}^{-1}(S) \cap \text{Writes}_x) ; \)
2. \( T \xrightarrow{\text{WW}_X(x)} S \iff T \xrightarrow{\text{AR}} S \land T, S \in \text{Writes}_x ; \)
3. \( T \xrightarrow{\text{RW}_X(x)} S \iff S \neq T \land (3T \xrightarrow{\text{WR}_X(x)} T \land T \xrightarrow{\text{WW}_X(x)} S ) . \)

Proposition 9. For any valid abstract execution \( \mathcal{X} \), \( \text{graph}(\mathcal{X}) \) is a dependency graph.

Specification of Consistency Models using Dependency Graphs. We interpret a dependency graph \( \mathcal{G} \) as a labelled graph whose vertices are transactions in \( T \), and whose edges are pairs of the form \( T \xrightarrow{R} S \), where \( R \in \{ \text{WR}_G(x), \text{WW}_G(x), \text{RW}_G(x) \mid x \in \text{Obj} \} \). To specify a consistency model, we employ a two-steps approach. We first identify one or more conditions to be satisfied by dependency graphs. Such conditions require cycles of a certain form not to appear in a dependency graph. Then we define a consistency model by projecting the set of dependency graphs satisfying the imposed conditions into the underlying histories. This style of specification is reminiscent of the one used in the CAT [4] language for formalising weak memory models. In the following we treat the relations \( \text{WR}_G(x), \text{WW}_G(x), \text{RW}_G(x) \) both as set-theoretic relations, and as edges of a labelled graph.

Definition 10. A dependency graph based specification, or simply g-specification, is a set \( \Delta = \{ \delta_1, \ldots, \delta_n \} \), where for each \( i \in \{1, \ldots, n\} \), \( \delta_i \) is a function of type \( \text{Graphs} \rightarrow 2^{(\mathbb{T} \times \mathbb{T})} \) and satisfies \( \delta_i(\mathcal{G}) \subseteq (\text{WR}_G \cup \text{WW}_G \cup \text{RW}_G)^* \) for every \( \mathcal{G} \in \text{Graphs} \).

Given a g-specification \( \Delta \), we define \( \text{Graphs}(\Delta) = \{ \mathcal{G} \in \text{Graphs} \mid \forall \delta \in \Delta, \delta(\mathcal{G}) \cap \text{Id} = \emptyset \} \), and we let \( \text{modelOf}(\Delta) = \{ T \mid \exists \mathcal{G}, \text{WR}, \text{WW}, \text{RW}, (T, \text{WR}, \text{WW}, \text{RW}) \in \text{Graphs}(\Delta) \} \).

The requirement imposed over the functions \( \delta_1, \ldots, \delta_n \) ensures that, whenever \( (T, S) \in \delta_i(\mathcal{G}) \), for some dependency graph \( \mathcal{G} \), then there exists a path in \( \mathcal{G} \) that connects \( T \) to \( S \). For \( \Delta = \{ \delta_i \}_{i=1}^n \) and \( \mathcal{G} \in \text{Graphs} \), the requirement that \( \delta_i(\mathcal{G}) \cap \text{Id} = \emptyset \) means that \( \mathcal{G} \) does not contain any cycle \( T_0 \xrightarrow{R_0} T_1 \xrightarrow{R_1} \cdots \xrightarrow{R_{n-1}} T_n \), such that \( T_0 = T_n \), and \( (R_0; \cdots; R_{n-1}) \subseteq \delta_i(\mathcal{G}) \).

Examples of g-specifications of consistency models. Below we give some examples of g-specifications for the consistency models presented in Section 2.

Theorem 11.

1. An execution \( \mathcal{X} \) is serialisable iff \( \text{graph}(\mathcal{X}) \) does not contain any cycle. That is, \( \text{modelOf}(\Sigma_{\text{SER}}) = \text{modelOf}(\{ \delta_{\text{SER}} \}) \), where \( \delta_{\text{SER}}(\mathcal{G}) = (\text{WR}_G \cup \text{WW}_G \cup \text{RW}_G)^* \).
2. An execution \( \mathcal{X} \) is allowed by snapshot isolation iff \( \text{graph}(\mathcal{X}) \) only admits cycles with at least two consecutive anti-dependency edge. That is, \( \text{modelOf}(\Sigma_{\text{SI}}) = \text{modelOf}(\{ \delta_{\text{SI}} \}) \), where \( \delta_{\text{SI}}(\mathcal{G}) = ((\text{WR}_G \cup \text{WW}_G) : \text{RW}_G)^* \).
3. An execution \( \mathcal{X} \) is allowed by parallel snapshot isolation iff \( \text{graph}(\mathcal{X}) \) has no cycle where all anti-dependency edges are over the same object. Let \( \delta_{\text{PSI}}(\mathcal{G}) = (\text{WR}_G \cup \text{WW}_G)^* \), \( \delta_{\text{PSI}(x)}(\mathcal{G}) = (\bigcup_{x \in \text{Obj}} (\text{WR}_G \cup \text{WW}_G)^* : \text{RW}_G(x))^* \), and define \( \Delta_{\text{PSI}} = \{ \delta_{\text{PSI}} \} \cup \{ \delta_{\text{PSI}(x)} \mid x \in \text{Obj} \} \). Then, \( \text{modelOf}(\Sigma_{\text{PSI}}) = \text{modelOf}(\Delta_{\text{PSI}}) \).

Theorem 11(1) was proved in [1]. The only if condition of Theorem 11(2) was proved in [19]; we proved the if condition of Theorem 11(2) in [16]. Theorem 11(3) improves on the specification we gave for PSI in [16]; the latter does not have any constraints on the objects to which anti-dependencies refer to. We outline the proof of Theorem 11(3) in Section 5.
Figure 3 Algebraic laws satisfied by an abstract execution \( X = (\mathcal{T}, \text{VIS}, \text{AR}) \). Here graph(\( X \)) = \((\mathcal{T}, \text{WR}, \text{WW}, \text{RW})\). The inequalities in part (d) are valid under the assumption that \( X \in \text{Executions}(\{(\rho, \pi)\}) \).

4 Algebraic Laws for Weak Consistency

Having two different styles for specifying consistency models gives rise to the following problems:

Weak Correspondence Problem. Given a \( X \)-specification \( \Sigma \), determine a non-trivial \( g \)-specification \( \Delta \) which over-approximates \( \Sigma \), that is such that \( \text{modelOf}(\Sigma) \subseteq \text{modelOf}(\Delta) \).

Strong Correspondence Problem. Given a \( X \)-specification \( \Sigma \), determine an equivalent \( g \)-specification \( \Delta \), that is such that \( \text{modelOf}(\Sigma) = \text{modelOf}(\Delta) \).

We first focus on the weak correspondence problem, and we discuss the strong correspondence problem in Section 5. This problem is not only of theoretical interest. Determining a \( g \)-specification \( \Delta \) that over-approximates a \( X \)-specification \( \Sigma \) corresponds to establishing one or more conditions satisfied by all cycles of dependency graphs from the set \{graph(\( X \)) \mid \( X \in \text{Executions}(\Sigma) \}\}. Cycles in a dependency graph that respect such a condition are called \( \Sigma \)-critical (or simply critical), and graphs that admit a non-\( \Sigma \)-critical cycle cannot be obtained from abstract executions in \( \text{Executions}(\Sigma) \). One can ensure that an application running under the model \( \Sigma \) is robust, i.e. it only produces serialisable behaviours, by checking for the absence of \( \Sigma \)-critical cycles at static time [7, 19]. Robustness of an application can also be checked at run-time, by incrementally constructing the dependency graph of executions, and detecting the presence of \( \Sigma \)-critical cycles [31].

General Methodology. Let \( \Sigma \) be a given \( X \)-specification. We tackle the weak correspondence problem in two steps.

First, we identify a set of inequalities that hold for all the executions \( X \) satisfying consistency guarantees \( (\rho, \pi) \) in \( \Sigma \). There are two kinds of such inequalities. The first are the inequalities in Figure 3, and the second the inequalities corresponding to the axioms of the Kleene Algebra \((2^{\mathcal{T} \times \mathcal{T}}, \emptyset, \cup, \cap, \cdot, *)\) and the Boolean algebra \((2^{\mathcal{T} \times \mathcal{T}}, \emptyset, \mathcal{T} \times \mathcal{T}, \cup, \cap, \cdot)\). The exact meaning of the inequalities in Figure 3 is discussed later in this section.
Second, we exploit our inequalities to derive other inequalities of the form $R_X \subseteq AR_X$ for every $X \in \text{Executions}(\Sigma)$. Here $R_X$ is a relation built from dependencies in graph($X$), i.e. $R_X \subseteq (WR_X \cup WW_X \cup RW_X)^*$. Because $AR_X$ is acyclic (that is $AR_X^+ \cap \text{ld} \subseteq \emptyset$), we may conclude that $R_X$ is acyclic for any $X \in \text{Executions}(\Sigma)$. In particular, we have that modelOf($\Sigma) \subseteq modelOf(\delta)$, where $\delta$ is a function that maps, for every abstract execution $X$, the dependency graph graph($X$) into the relation $R_X$.

Some of the inequalities we develop, namely those in Figure 3(d), are parametric in the consistency guarantee $(\rho, \pi)$. As a consequence, our approach can be specialised to any consistency model that is captured by our framework. To show its applicability, we derive critical cycles for several of the consistency models that we have presented.

**Presentation of the Laws.** Let $X = (T, VIS, AR)$, and graph($X$) = (T, WR, WW, RW). We now explain the inequalities in Figure 3. Among these, the inequalities in Figures 3(a) and (b) should be self-explanatory.

Let us discuss the inequalities of Figure 3(c). The inequalities (c.1), (c.2) and (c.3) relate dependencies to either basic or derived relations of abstract executions. Dependencies of the form WR, WW are included in the relations VIS, AR, respectively, as established by inequalities (c.1) and (c.2). The inequality (c.3), which we prove presently, is non-standard. It relates anti-dependencies to a novel anti-visibility relation VIS-, defined as $T \xrightarrow{VIS^-} S$ iff $\neg(S \xrightarrow{VIS} T)$. In words, $S$ is anti-visible to $T$ if $T$ does not observe the effects of $S$. As we will explain later, anti-visibility plays a fundamental role in the development of our laws.

**Proof of Inequality (c.3).** Suppose $T \xrightarrow{RW(x)} S$ for some object $x \in \text{Obj}$. By definition, $T \neq S$, and there exists a transaction $T'$ such that $T' \xrightarrow{WR(x)} T$ and $T' \xrightarrow{WW(x)} S$. In particular, $T' \xrightarrow{VIS} T$ and $T' \xrightarrow{AR} S$ by the inequalities (c.1) and (c.2), respectively. Now, if it were $S \xrightarrow{VIS} T$, then we would have that $T'$ is not the AR-supremum of the set of transactions visible to $T$, and writing to object $x$. But this contradicts the definition of graph($X$), and the edge $T' \xrightarrow{WR(x)} T$. Therefore, $T \xrightarrow{VIS^-} S$.

Another non-trivial inequality is (c.7) in Figure 3(c). It says that if a transaction $T$ reads a value for an object $x$ that is later updated by another transaction $S$ ($T \xrightarrow{RW} S$), then the update of $S$ is more recent (i.e. it follows in arbitration) than all the updates to $x$ seen by $T$. We prove it in [17, Appendix C]. The other inequalities in Figure 3(c) are self explanatory.

The inequalities in Figure 3(d) are specific to a consistency guarantee $(\rho, \pi)$, and hold for an execution $X$ when the execution satisfies $(\rho, \pi)$. The inequality (d.1) is just the definition of consistency guarantee. The next inequality (d.2) is where the novel anti-visibility relation, introduced previously, comes into play. While the consistency guarantee $(\rho, \pi)$ expresses when arbitration induces transactions related by visibility, the inequality (d.2) expresses when anti-visibility induces transactions related by arbitration. To emphasise this correspondence, we call the inequality (d.2) co-axiom induced by $(\rho, \pi)$. Later in this section, we show how by exploiting the co-axiom induced by several consistency guarantees, we can derive critical cycles of several consistency models.

**Proof of Inequality (d.2).** Assume $X \in \text{Executions}((\rho, \pi))$. Let $T, T', S', S \in T$ be such that $T \neq S$, $T \xrightarrow{\rho(VIS)} T' \xrightarrow{VIS} S' \xrightarrow{\rho(VIS)} S$. Because AR is total, either $S \xrightarrow{AR} T$ or $T \xrightarrow{AR} S$. However, the former case is not possible. If so, we would have $S' \xrightarrow{\rho(VIS)} S \xrightarrow{AR} T \xrightarrow{VIS} T'$.
because $\mathcal{X} \in \text{Executions}((\rho, \pi))$, by the inequality \((d.1)\), it would follow that $S' \overset{\text{VIS}}{\rightarrow} T'$, contradicting the assumption that $T' \overset{\text{AR}}{\rightarrow} S$. Therefore, it has to be $T \overset{\text{AR}}{\rightarrow} S$. ◀

The last inequalities \((d.3)\) and \((d.4)\) in Figure 3(d) show that anti-visibility edges of $\mathcal{X}$ are also induced by the consistency guarantee $p, \pi, q$. We prove them formally in [17, Appendix C], where we also illustrate some of their applications.

**Applications.** We employ the algebraic laws of Figure 3 to derive $\Sigma$-critical cycles for arbitrary $x$-specifications, using the methodology explained previously: given a $x$-specification $\Sigma$ and an abstract execution $\mathcal{X}$, we characterise a subset of $\text{AR}_x$ as a relation $\mathcal{R}_G$ built from the dependencies in graph $p, \pi, q$ and relations of the form $[o]$, where $o \in \text{Op}$. Because $\mathcal{R}_G \subseteq \text{AR}_x$, we conclude that $\mathcal{R}_G$ is acyclic.

The inequalities \((c.1)\), \((c.6)\) and \((c.2)\) ensure that we can always include write-read and write-write dependencies in the relation $\mathcal{R}_G$ above. Because of inequalities \((c.3)\) and \((d.2)\) (among others), we can include in $\mathcal{R}_G$ also relations that involve anti-dependencies. The following result shows how this methodology can be applied to serialisability. We use the notation $\mathcal{R}_1 \overset{(eq)}{\subseteq} \mathcal{R}_2$ to denote that the inequality $\mathcal{R}_1 \subseteq \mathcal{R}_2$ follows from \((eq)\).

**Theorem 12.** For all $\mathcal{X} \in \text{Executions}(\Sigma_{\text{SER}})$, the relation $(\text{WR}_X \cup \text{WW}_X \cup \text{RW}_X)$ is acyclic.

**Proof.** Recall that $\Sigma_{\text{SER}} = \{(\rho_{\text{a}}, \rho_{\text{a}})\}$, where $\rho_{\text{a}}(\_\_\_\_) = \text{Id}$. We have

\[
\text{RW}_X \overset{(b.6)}{\subseteq} \text{RW}_X \setminus \text{Id} \overset{(c.3)}{\subseteq} \text{VIS}_X \setminus \text{Id} = (\rho_{\text{a}}(\text{VIS}_X) ; \text{VIS}_X) \setminus \text{Id} \overset{(d.2)}{\subseteq} \text{AR}_X \tag{1}
\]

\[
(\text{WR}_X \cup \text{WW}_X \cup \text{RW}_X) \setminus \text{Id} \overset{(c.1, c.6)}{\subseteq} (\text{AR}_X \cup \text{WW}_X \cup \text{RW}_X) \setminus \text{Id} \overset{(c.2)}{\subseteq} (\text{AR}_X \cup \text{WR}_X) \setminus \text{Id} \overset{(1)}{\subseteq} \text{AR}_X \tag{2}
\]

\[
(\text{WR}_X \cup \text{WW}_X \cup \text{RW}_X)^+ \cap \text{Id} \overset{(2)}{\subseteq} \text{AR}_X^+ \cap \text{Id} \overset{(c.5)}{\subseteq} \text{AR}_X \cap \text{Id} \overset{(c.12)}{\subseteq} \emptyset. \quad \blacktriangleleft
\]

Along the lines of the proof of Theorem 12, we can characterise $\Sigma$-critical cycles for an arbitrary $x$-specification $\Sigma$. Below, we show how to apply our methodology to derive $\Sigma_{\text{RB}}$-critical cycles.

**Theorem 13.** Let $\mathcal{X} \in \text{Executions}(\Sigma_{\text{RB}})$. Say that a $\text{RW}_X$ edge in a cycle of graph($\mathcal{X}$) is protected if its endpoints are connected to serialisable transactions via a sequence of $\text{WR}_X$ edges. Then all cycles in graph($\mathcal{X}$) have at least one unprotected $\text{RW}_X$ edge. Formally, let $\mathcal{R}^-\text{RW}_X$ be $([\text{SerTx}] ; (\text{WR}_X)^* ; \text{RW}_X ; (\text{WR}_X)^* ; [\text{SerTx}])$. Then $(\text{WR}_X \cup \text{WW}_X \cup \mathcal{R}^-\text{RW}_X)$ is acyclic.
We now turn our attention to the x-specifications. This class includes several of the consistency models we have presented.

**Proof.** It suffices to prove that \( \vdash \text{RW}_\mathcal{X} \models \subseteq \text{AR}_\mathcal{X} \). The rest of the proof is similar to the one of Theorem 12. We recall that \( \Sigma_{\text{RB}} = \{(\rho_S, \rho_S)\} \), where \( \rho_S(\_\_) = [\text{Ser} \_\_]. \)

\[
\begin{align*}
\text{WR} & \subseteq X_V \quad \text{(V1)} \quad X_V : X_V \subseteq X_V \quad \text{(V2)} \quad \bigcup_{\{x \mid (\rho_x, \rho_x) \in \Sigma\}} \text{WW}(x) \subseteq X_V \quad \text{(V3)} \\
\text{WW} & \subseteq X_A \quad \text{(A1)} \quad X_V \subseteq X_A \quad \text{(A2)} \quad \bigcup_{x \in \text{Obj}} ([\text{Writes}_{x}] ; X_V \; \text{RW}(x)) \subseteq X_A \quad \text{(A3)} \\
\text{RW} & \subseteq X_N \quad \text{(N1)} \quad X_V : X_N \subseteq X_N \quad \text{(N2)} \quad \pi(X_V) ; X_N \; \rho(X_V) \subseteq X_A \quad \text{(N3)}
\end{align*}
\]

\[\text{Figure 4} \] The system of inequalities \( \text{System}_{\Sigma}(\mathcal{G}) \) for the simple consistency model \( \Sigma \) and the dependency graph \( \mathcal{G} = (\mathcal{T}, \text{WR, WW, RW}) \).

We remark that our characterisation of \( \Sigma_{\text{RB}} \)-critical cycle cannot be compared to the one given in [7]. In [17, Appendix C] we show how our methodology can be applied to give a characterisation of \( \Sigma_{\text{RB}} \)-critical cycles that is stronger than both the one presented in Theorem 13 and the one given in [7]. We also employ our proof technique to prove both known and new derivations of critical cycles for other x-specifications.

## 5 Characterisation of Simple Consistency Models

We now turn our attention to the Strong Correspondence Problem presented in Section 4. Given a x-specification \( \Sigma = \{(\rho_1, \pi_1), \cdots, (\rho_n, \pi_n)\} \) and a dependency graph \( \mathcal{G} \), we want to find a sufficient and necessary condition for determining whether \( \mathcal{G} = \text{graph}(\mathcal{X}) \) for some \( \mathcal{X} \in \text{Executions}(\Sigma) \).

In this section we propose a proof technique for solving the strong correspondence problem. This technique applies to a particular class of x-specifications, which we call simple x-specifications. This class includes several of the consistency models we have presented.

### Characterisation of Simple x-specifications

Recall that for each \( x \in \text{Obj} \), the function \( \rho_x \) of an abstract execution \( \mathcal{X} \) is defined as \( \rho_x(\_\_) = [\text{Writes}_{x}] \), and the associated axiom is \( [\text{Writes}_{x}] ; \text{AR}_{\mathcal{X}} \subseteq \text{VIS}_{\mathcal{X}} \).
We define a system of inequalities $\Sigma$ containing at most one consistency guarantee, beside those of the form $(\rho_x, \rho_y)$ which express the write-conflict detection for some object $x \in \text{Obj}$. Among the x specifications that we have presented in this paper, the only non-simple one is $\Sigma_{SI+SER}$.

For simple x specifications, it is possible to solve the strong correspondence problem. Fix a simple x specification $\Sigma \subseteq \{(\rho, \pi)\} \cup \{(\rho_x, \rho_y) \mid x \in \text{Obj}\}$ and a dependency graph $G$. We define a system of inequalities $\text{System}_\Sigma(G)$ in three unknowns $X_V, X_A,$ and $X_N,$ and depicted in Figure 4 (the inequalities (V4) and (A5) are included in the system if and only if $(\rho, \pi) \in \Sigma$). These unknowns correspond to subsets of the visibility, arbitration and anti-visibility relations of the abstract execution $\mathcal{X} \in \text{Executions}(\Sigma)$, with underlying dependency graph $G$, that we wish to find. Note that each one of the inequalities of $\text{System}_\Sigma(G)$, with the exception of (V3), follows the structure of one of the algebraic laws from Figure 3. We prove that, in order to ensure that the abstract execution $\mathcal{X}$ exists, it is sufficient to find a solution of $\text{System}_\Sigma(G)$ whose $X_A$ component is acyclic. In particular, this is true if and only if the $X_A$ component of the smallest solution of $\text{System}_\Sigma(G)$ is acyclic.

\textbf{Theorem 15.}

\textbf{Soundness:} for any $\mathcal{X} \in \text{Executions}(\Sigma)$ such that $\text{graph}(\mathcal{X}) = G$, the triple $(X_V = V\text{IS}_\mathcal{X}, X_A = AR_\mathcal{X}, X_N = V\text{IS}_\mathcal{X}^{-1})$ is a solution of $\text{System}_\Sigma(G)$.

\textbf{Completeness:} Let $(X_V = V\text{IS}_0, X_A = AR_0, X_N = \text{AntiVIS}_0)$ be the smallest solution of $\text{System}_\Sigma(G)$. If $AR_0$ is acyclic, then there exists an abstract execution $\mathcal{X}$ such that $\mathcal{X} \in \text{Executions}(\Sigma)$ and $\text{graph}(\mathcal{X}) = G$. ▶

Note that the relation $AR_0$ need not to be total in the completeness direction of Theorem 15.

Before discussing the proof of Theorem 15, we show how it can be used to prove the equivalence of a x specification and a g specification. We give a proof of Theorem 11(3). Theorems 11(1) and 11(2) can be proved similarly, and their proof is given in [17, Appendix D].

\textbf{Proof Sketch of Theorem 11(3).} Recall that $\Delta_{PSI} = \delta_{PSI(1)} \cup \delta_{PSI(2)}(G) \mid x \in \text{Obj}$, where $\delta_{PSI(1)}(G) = (WR_{G} \cup WW_{G})^+, \delta_{PSI(2)}(G) = ((WR_G \cup WW_{G}* : RW_{G}(x))^+$. In [17, Appendix D] we prove that $\text{Graphs}(\Delta_{PSI}) = \text{Graphs}(\{\delta_{PSI}\})$, where

$$\delta_{PSI}(G) = (WR_G \cup WW_{G})^+ \cup \bigcup_{x \in \text{Obj}} (WR_G \cup WW_{G})* : RW_{G}(x))^+.$$ 

Therefore, it suffices to prove that $\text{modelOf}(\Sigma_{PSI}) = \text{modelOf}(\{\delta_{PSI}\})$:

\textbf{modelOf}(\Sigma_{PSI}) \subseteq \text{modelOf}(\{\delta_{PSI}\}): given $\mathcal{X} \in \text{Executions}(\Sigma_{PSI})$, and let $G := \text{graph}(\mathcal{X})$, we need to show that $\delta_{PSI}(G) \cap \text{Id} = G$. The proof follows the style of Theorems 12 and 13: details can be found in [17, Appendix C].

\textbf{modelOf}(\{\delta_{PSI}\}) \subseteq \text{modelOf}(\Sigma_{PSI}): given $G \in \text{Graphs}(\delta_{PSI})$, let $V\text{IS}_G = (WR \cup WW)^+; V\text{IS}_G^-$ is a solution of $\text{System}_\Sigma(G)$. Because $\delta_{PSI}(G)$ is acyclic, if we take the smallest solution $(X_V = \_ , X_A = AR_G, X_N = \_)$ of $\text{System}_\Sigma(G)$, then $AR_G \subseteq \delta_{PSI}(G)$, hence $AR_G$ is acyclic. By Theorem 15, there exists an abstract execution $\mathcal{X} \in \text{Executions}(\Sigma)$ such that $\text{graph}(\mathcal{X}) = G$, and in particular $T_X = T_G$. ▶

\textbf{Definition 14.} A x specification $\Sigma$ is \emph{simple} if there exists a consistency guarantee $(\rho, \pi)$ such that $\Sigma \subseteq \{(\rho, \pi)\} \cup \{(\rho_x, \rho_y) \mid x \in \text{Obj}\}$. A solution $(X_V = VIS, X_A = AR, X_N = \text{AntiVIS})$ is smaller than another one $(X_V = VIS', X_A = AR', X_N = \text{AntiVIS}')$ if $VIS \subseteq VIS', AR \subseteq AR'$ and $\text{AntiVIS} \subseteq \text{AntiVIS}'$.\footnote{A solution $(X_V = VIS, X_A = AR, X_N = \text{AntiVIS})$ is smaller than another one $(X_V = VIS', X_A = AR', X_N = \text{AntiVIS}')$ if $VIS \subseteq VIS', AR \subseteq AR'$ and $\text{AntiVIS} \subseteq \text{AntiVIS}'$.}
We now turn our attention to the proof of Theorem 15. The proof of the soundness direction is straightforward.

**Proof of Theorem 15 (Soundness).** Let \( X \in \text{Executions}(\Sigma) \), and define \( \mathcal{G} := \text{graph}(X) \). To show that the triple \((X_V = \text{VIS}_X, X_A = \text{AR}_X, X_N = \overline{\text{VIS}_X})\) is a solution of \( \text{System}_\Sigma(\mathcal{G}) \), we need to show that all the inequalities from said system are satisfied, when the unknowns \( X_A, X_V, X_N \) are replaced with \( \text{VIS}_X, \text{AR}_X, \overline{\text{VIS}_X} \), respectively. In practice, all the inequalities, with the exception of \((V3)\), follow from the algebraic laws of Figure 3. Let us prove that \((V3)\) is also valid: for any \((\rho_x, \rho_z) \in \Sigma\) we have that

\[
\text{WW}_X(x) \stackrel{\text{b.2}}{=} [\text{Writes}_x] ; \text{WW}_X(x) ; [\text{Writes}_x] \subseteq [\text{Writes}_x] ; \text{AR}_X ; [\text{Writes}_x] \stackrel{\text{d.1}}{=} \text{VIS}_X.
\]

The proof of the completeness direction of Theorem 15 is much less straightforward. Let \((X_V = \text{VIS}_0, X_A = \text{AR}_0, X_N = \text{AntiVIS}_0)\) be the smallest solution of \( \text{System}_\Sigma(\mathcal{G}) \). Assume that \( \text{AR}_0 \) is acyclic. The challenge is that of constructing a valid abstract execution \( X \), i.e. whose arbitration order is total, from the dependencies in \( \mathcal{G} \), that is included in \( \text{Executions}(\Sigma) \). We do this incrementally: at intermediate stages of the construction we get structures similar to abstract executions, but where the arbitration order can be partial.

**Definition 16.** A pre-execution \( P = (T_\mathcal{G}, \text{VIS}, \text{AR}) \) is a tuple that satisfies all the constraints of abstract executions, except that \( \text{AR} \) is not necessarily total, although \( \text{AR} \) is still required to be total over the set \( \text{Writes}_x \) for every object \( x \).

The notation adopted for abstract executions naturally extends to pre-executions; also, for any pre-execution \( P \), \( \text{graph}(P) \) is a well-defined dependency graph. Given a \( \Sigma \)-specification \( \Sigma \), we let \( \text{PreExecutions}(\Sigma) \) be the set of all valid pre-executions that satisfy all the consistency guarantees in \( \Sigma \).

\( \text{System}_\Sigma(\mathcal{G}) \) is defined so that all of its solutions whose \( X_A \)-component is acyclic induce a valid pre-execution in \( \text{PreExecutions}(\Sigma) \) with underlying dependency graph \( \mathcal{G} \).

**Proposition 17.** Let \((X_V = \text{VIS}', X_A = \text{AR}', X_N = \text{AntiVIS}')\) be a solution to \( \text{System}_\Sigma(\mathcal{G}) \). If \( \text{AR}' \cap \text{Id} = \emptyset \), then \( P = (T_\mathcal{G}, \text{VIS}', \text{AR}') \in \text{PreExecutions}(\Sigma) \); moreover, \( \text{graph}(P) = \mathcal{G} \).

**Proof Sketch.** The inequalities \((A1), (A2)\) and \((A4)\) together with the assumption that \( \text{AR}_0 \) is acyclic, ensure that \( P \) is a pre-execution. In particular, \((A1)\) ensures that \( \text{AR}_0 \) is a total relation over the set \( \text{Writes}_x \), for any \( x \in \text{Obj} \). As we explain in [17, Appendix D], the inequalities \((V1), (A1)\) and \((A3)\) enforce the Last Write Wins policy (Definition 3). The inequality \((V2)\) mandates that \( P \) respects causality. Finally, the inequalities \((V3)\) and \((V4)\) ensure that all the consistency guarantees in \( \Sigma \) are satisfied by \( P \).}

In particular, the smallest solution \((X_V = \text{VIS}_0, X_A = \text{AR}_0, X_N = \text{AntiVIS}_0)\) of \( \text{System}_\Sigma(\mathcal{G}) \) induces the pre-execution \((T_\mathcal{G}, \text{VIS}_0, \text{AR}_0) \in \text{PreExecutions}(\Sigma) \).

To construct an abstract execution \( X \in \text{Executions}(\Sigma) \), with \( \text{graph}(X) = \mathcal{G} \), we define a finite chain of pre-executions \( \{P_i\}_{i=0}^n \), \( n \geq 0 \), as follows: (i) let \( P_0 := (T_\mathcal{G}, \text{VIS}_0, \text{AR}_0) \); (ii) given \( P_i, i \geq 0 \), choose two different transactions \( T_i, S_i \in T_\mathcal{G} \) (if any) that are not related by \( \text{AR}_i \), compute the smallest solution \((X_V = \text{VIS}_{i+1}, X_A = \text{AR}_{i+1}, X_N = \ldots)\) such that \( \text{AR}_{i+1} \supseteq \text{AR}_i \cup \{(T_i, S_i)\} \), and let \( P_{i+1} := (T_\mathcal{G}, \text{VIS}_{i+1}, \text{AR}_{i+1}) \); (iii) if the transactions \( T_i, S_i \in T_\mathcal{G} \) from the previous step do not exist, then let \( n := i \) and terminate the construction. Because we are assuming that \( T_\mathcal{G} \) is finite, the construction of \( \{P_0, \ldots, P_n\} \) always terminates.
we have shown that, for a particular class of consistency models, our algebraic approach leads to a precise characterisation of consistency models in terms of dependency graphs.

**Proposition 18.** For \( i = 0, \ldots, n - 1 \), let \( \Delta R_i := R_i \cup \{ (T_i, S_i) \} \cup R_i \). Then \( R_{i+1} = R_i \cup \Delta R_i \).

**Corollary 19.** For \( i = 0, \ldots, n - 1 \), if \( R_i \cap Id = \emptyset \), then \( R_{i+1} \cap Id = \emptyset \).

**Proof.** Because \( R_i \cap Id = \emptyset \) by hypothesis, by Proposition 18 we only need to show that \( \Delta R_i \cap Id = \emptyset \). If \((T, T) \in \Delta R_i \) for some \( T \in \mathcal{T}_G \), then it must be \( T \xrightarrow{AR_i \cup \Delta R_i} T \) and \( S_i \xrightarrow{AR_i \cup \Delta R_i} T \). It follows that \( S_i \xrightarrow{AR_i \cup \Delta R_i} T_i \). But this contradicts the hypothesis that \( R_i \) does not relate transactions \( T_i \) and \( S_i \). Therefore, \((T, T) \notin \Delta R_i \) for any \( T \in \mathcal{T}_G \), i.e. \( \Delta R_i \cap Id = \emptyset \).

We have now everything in place to prove Theorem 15.

**Proof of Theorem 15 (Completeness).** Let \( \mathcal{G} \) be a dependency graph, and define the chain of pre-executions \( P_0 = (\mathcal{T}_G, \text{VIS}_0, AR_0), \ldots, P_n = (\mathcal{T}_G, \text{VIS}_n, AR_n) \) as described above. We show that for any \( i = 0, \ldots, n \), \( P_i \in \text{PreExecutions}(\Sigma) \), and \( \text{graph}(P_i) = \mathcal{G} \). Because \( AR_n \) is a total order, this implies that \( P_n \in \text{Executions}(\Sigma) \), and \( \text{graph}(P_n) = \mathcal{G} \), as we wanted to prove. The proof is by induction on \( n \).

**Case** \( i = 0 \): observe that the triple \((X_V = \text{VIS}_0, X_A = AR_0, X_N = _)\) corresponds to the smallest solution of \( \text{System}_{\Sigma}(\mathcal{G}) \), hence \( AR_0 \) is acyclic by hypothesis. It follows from Proposition 17 that \( P_0 \in \text{PreExecutions}(\Sigma) \), and \( \text{graph}(P_0) = \mathcal{G} \).

**Case** \( i > 0 \): assume that \( i < n \); then \( i - 1 < n \), and by induction hypothesis \( P_{i-1} \in \text{PreExecutions}(\Sigma) \). In particular, the relation \( AR_{i-1} \) is acyclic; by Corollary 19 we obtain that \( AR_i \) is acyclic. Finally, recall that the triple \((X_V = \text{VIS}_i, X_A = AR_i, X_N = _)\) is a solution of \( \text{System}_{\Sigma}(\mathcal{G}) \) by construction. It follows from Proposition 17 that \( P_i \in \text{PreExecutions}(\Sigma) \), and \( \text{graph}(P_i) = \mathcal{G} \).

**6 Conclusion**

We have explored the connection between two different styles of specifications for weak consistency models at an algebraic level. We have proposed several laws which we applied to devise several robustness criteria for consistency models. To the best of our knowledge, this is the first generic proof technique for proving robustness criteria of weak consistency models. We have shown that, for a particular class of consistency models, our algebraic approach leads to a precise characterisation of consistency models in terms of dependency graphs.

**Related Work.** Abstract executions have been introduced by Burckhardt in [12] to model the behaviour of eventually consistent data-stores; They have been used to capture the behaviour of replicated data types (Gotsman et al. [13]), geo-replicated databases (Cerone et al. [15]) and non-transactional distributed storage systems (Viotti et al. [30]).
Dependency graphs have been introduced by Adya [1]; they have been used since to reason about programs running under weak consistency models. Bernardi et al., used dependency graphs to derive robustness criteria of several consistency models [7], including PSI and red-blue; in contrast with our work, the proofs there contained do not rely on a general technique. Brutschy et al. generalised the notion of dependency graphs to replicated data types, and proposed a robustness criterion for eventual consistency [10].

Weak consistency also arises in the context of shared memory systems [4]. Alglave et al., proposed the CAT language for specifying weak memory models in [4], which also specifies weak memory models as a set of irreflexive relations over data-dependencies of executions. Castellan [14], and Jeffrey et al. [21], proposed different formalisations of weak memory models via event structures. The problem of checking the robustness of applications has also been addressed for weak memory models [2, 3, 8].

The strong correspondence problem (Section 5) is also highlighted by Bouajjani et al. in [9]: there the authors emphasize the need for general techniques to identify all the bad patterns that can arise in dependency-graphs like structures. We solved the strong correspondence problem for SI in [16].

References


29 Douglas B Terry, Alan J Demers, Karin Petersen, Mike J Spreitzer, Marvin M Theimer, and Brent B Welch. Session guarantees for weakly consistent replicated data.
Algebraic Laws for Weak Consistency


Algorithms to Compute Probabilistic Bisimilarity Distances for Labelled Markov Chains

Qiyi Tang\textsuperscript{1} and Franck van Breugel\textsuperscript{2}

\textsuperscript{1} DisCoVeri Group, Department of Electrical Engineering and Computer Science, York University, Toronto
\textsuperscript{2} DisCoVeri Group, Department of Electrical Engineering and Computer Science, York University, Toronto

Abstract
In the late nineties, Desharnais, Gupta, Jagadeesan and Panangaden presented probabilistic bisimilarity distances on the states of a labelled Markov chain. This provided a quantitative generalisation of probabilistic bisimilarity introduced by Larsen and Skou a decade earlier. In the last decade, several algorithms to approximate and compute these probabilistic bisimilarity distances have been put forward. In this paper, we correct, improve and generalise some of these algorithms. Furthermore, we compare their performance experimentally.

1998 ACM Subject Classification D.2.4 Software/Program Verification, F.1.1 Models of Computation, G.3 Probability and Statistics

Keywords and phrases labelled Markov chain, probabilistic bisimilarity, pseudometric, policy iteration

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2017.27

1 Introduction

For the last three and a half decades behavioural equivalences such as bisimilarity, due to Milner [22] and Park [25], have been a cornerstone of concurrency theory. As this theory matured, more detailed models of concurrent systems have been developed, such as models with probabilities. For these enriched models, behavioural equivalences were proposed, such as probabilistic bisimilarity due to Larsen and Skou [21].

Although these behavioural equivalences allow us to reason about the behaviour of these models, they have one drawback: they are not robust. That is, small changes in the probabilities may cause equivalent states to becomes inequivalent or vice versa. Since the probabilities are usually obtained experimentally and, therefore are often just an approximation, this lack of robustness is a serious limitation of behavioural equivalences for these models with probabilities. This lack of robustness was first pointed out by Giacalone, Jou and Smolka [13]. They suggested generalising behavioural equivalences, which assign to each pair of states a Boolean, to behavioural pseudometrics, which assign to each pair of states a nonnegative real number.

The distance of a pair of states provides a quantitative measure of their behavioural similarity. The smaller this distance, the more alike the states behave. Distance zero captures that the states behave exactly the same, that is, they are behaviourally equivalent.
For a more detailed discussion of the merits of behavioural pseudometrics, we refer the reader to, for example, [24, Chapter 8].

Labelled Markov chains are often used to model systems with probabilistic behaviour. An example of such a Markov chain is depicted below. In a labelled Markov chain, each state has a label. In the example below, the label is represented by the colour of the state. These labels are used to capture that particular properties of interest hold in some states and do not hold in other states.

Several behavioural pseudometrics for labelled Markov chains have been proposed, the probabilistic bisimilarity pseudometric due to Desharnais, Gupta, Jagadeesan and Panangaden [12] being the most notable one. In this paper, we focus on this probabilistic bisimilarity pseudometric. Some of the probabilistic bisimilarity distances for the above labelled Markov chain can be found in the table below. For some historical background on this behavioural pseudometric we refer the reader to [6, Section 1].

<table>
<thead>
<tr>
<th></th>
<th>s</th>
<th>t</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>0</td>
<td>1/2</td>
<td>3/4</td>
<td>3/4</td>
<td>1</td>
<td>3/4</td>
</tr>
<tr>
<td>t</td>
<td>1/2</td>
<td>0</td>
<td>3/4</td>
<td>3/4</td>
<td>1</td>
<td>3/4</td>
</tr>
<tr>
<td>a</td>
<td>3/4</td>
<td>3/4</td>
<td>0</td>
<td>1/2</td>
<td>1</td>
<td>1/2</td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1/2</td>
<td>1/2</td>
</tr>
<tr>
<td>c</td>
<td>1</td>
<td>1</td>
<td>1/2</td>
<td>0</td>
<td>1/2</td>
<td>1/2</td>
</tr>
<tr>
<td>d</td>
<td>1/2</td>
<td>3/4</td>
<td>1/2</td>
<td>1/2</td>
<td>1/2</td>
<td>0</td>
</tr>
</tbody>
</table>

In order to exploit behavioural pseudometrics such as the probabilistic bisimilarity pseudometric, it is essential to be able to approximate or compute these behavioural distances. The first algorithm to approximate these distances was presented by Van Breugel, Sharma and Worrell in [5]. In their algorithm, the distance between states s and t, denoted $\delta(s,t)$, is computed as follows. Since $\delta(s,t) < q$, for some rational $q$, can be expressed in the existential fragment of the first order theory over the reals as shown by Van Breugel et al., and this theory is decidable as shown by Tarski [29], one can use binary search to approximate $\delta(s,t)$. The satisfiability problem for the existential fragment of the first order theory over the reals can be solved in polynomial space [7].

Subsequently, Chen, Van Breugel and Worrell [8] presented a polynomial time algorithm to compute the distances. They showed that the distances are rational and that those distances can be computed by means of Khachiyan’s ellipsoid method [17]. In particular, they showed that the distance function can be expressed as the solution of a linear program. In this case, the separation algorithm, which is an integral part of the ellipsoid method, boils down to solving a minimum cost flow problem. The network simplex algorithm solves the latter problem in polynomial time [23].
It is well known that the ellipsoid method is simpler when the feasible region is bounded and full-dimensional (see, for example, [26, Chapter 13]). In Section 4 we show that the feasible region of the linear program used by Chen et al. to compute the probabilistic bisimilarity distances is bounded and full-dimensional. As a consequence, we can use the simpler version of the ellipsoid method to compute the probabilistic bisimilarity distances.

Bacci, Bacci, Larsen and Mardare [1] put forward yet another algorithm to compute the probabilistic bisimilarity distances. Their algorithm can be viewed as a basic algorithm, enhanced with an optimization. The key idea behind this optimization is to compute the distances on-the-fly. We will come back to this optimization later, but we will first focus on the basic algorithm.

In [28], we show that a slight modification of the basic algorithm of Bacci et al. can be seen as an incarnation of simple policy iteration, also known as sequential policy iteration. In particular, we construct a transformation mapping each labelled Markov chain to a simple stochastic game, a simplification of Shapley’s stochastic games [27] due to Condon [9]. The vertices of the simple stochastic game represent pairs of states of the labelled Markov chain. Furthermore, the probabilistic bisimilarity distance of two states of the labelled Markov chain is shown to be equal to the value of the corresponding vertex of the simple stochastic game.

A variety of algorithms has been developed to compute the value function of a simple stochastic game. Several of these algorithms use policy iteration. As long as there exists a choice in the strategy, also known as a policy, of a player that is not locally optimal, switch that choice for one that is locally optimal. Hoffman and Karp [15] introduced a policy iteration algorithm for stochastic games in which all non-optimal choices are switched in each iteration. Condon [10] presented a similar algorithm, known as simple policy iteration, that switches only one non-optimal choice per iteration.

In Section 5 we present our simple policy iteration algorithm. Furthermore, we modify this algorithm to mimic general policy iteration a la Karp and Hoffman. We prove both algorithms correct. In Section 6 we consider the on-the-fly optimization due to Bacci et al. [1]. This optimization boils down to considering partial policies. That is, a player only makes choices for an appropriate subset of the vertices. We call it partial policy iteration. As we will show, Bacci et al. do not always consider partial policies that are defined for sufficiently many vertices. We propose a modification of their algorithm and prove it correct. A partial variant that mimics general policy iteration can be developed and proved correct similarly.

In [28], we show that the number of iterations of the simple policy iteration algorithm is at least exponential in the number of states of the labelled Markov chain. In Section 7 we prove that this is also the case for the simple partial policy iteration algorithm, even if we are only interested in the distance of a single pair of states.

The algorithms considered in this paper are the one which applies the first order theory over the reals, the ellipsoid method, simple policy iteration, general policy iteration, simple partial policy iteration and general partial policy iteration. To compare the performance of these six algorithms to approximate and compute probabilistic bisimilarity distances for labelled Markov chain, we ran several experiments. All six algorithms were implemented in Java. These implementations were run on a number of labelled Markov chains, which model well-known randomized algorithms and were obtained from examples of probabilistic model checkers such as PRISM [20] and jpf-probabilistic [31]. These experiments and the results are discussed in Section 8.

The contributions of this paper are the following. In Section 4 we show that the feasible region of the linear programming of Chen et al. describing the probabilistic bisimilarity dis-
Probabilistic Bisimilarity Distances

As a consequence, we can resort to a simpler version of the ellipsoid method to compute the distances. We study the on-the-fly optimization of the algorithm of Bacci et al. in Section 6 and show that it does not always consider sufficiently many states. We modify their optimization and prove our modification correct. We consider this our main contribution. In Section 7 we prove an exponential lower bound for the simple partial policy iteration algorithm. Finally, by means of experiments we compare the performance of the different algorithms to approximate and compute the probabilistic bisimilarity distances in Section 8.

2 Labelled Markov Chains and Probabilistic Bisimilarity

We start by reviewing the model of interest, labelled Markov chains, and its most well known behavioural equivalence, probabilistic bisimilarity due to Larsen and Skou [21]. We denote the set of probability distributions on a finite set $S$ by $\text{Distr}(S)$.

Definition 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 function $\tau : S \to \text{Distr}(S)$, and
- a labelling function $\ell : S \to L$.

We restrict our attention to labelled Markov chains with rational transition probabilities. For the remainder of this paper, we fix such a labelled Markov chain. In order to characterize probabilistic bisimilarity, we first introduce the notion of a coupling of probability distributions.

Definition 2. Let $\mu, \nu \in \text{Distr}(S)$. The set $\Omega(\mu, \nu)$ of couplings $\omega$ of $\mu$ and $\nu$ is defined by
\[
\Omega(\mu, \nu) = \left\{ \omega \in \text{Distr}(S \times S) \mid \sum_{t \in S} \omega(s, t) = \mu(s) \land \sum_{s \in S} \omega(s, t) = \nu(t) \right\}.
\]

The set $\Omega(\mu, \nu)$ is a convex polytope. We denote its vertices by $V(\Omega(\mu, \nu))$. Generally, the set $\Omega(\mu, \nu)$ is infinite, but the set $V(\Omega(\mu, \nu))$ is finite (see, for example, [18, page 259]).

The following characterization of probabilistic bisimilarity, in terms of couplings, is due to Jonsson and Larsen [16, Theorem 4.6]. For a discussion of the notion of a coupling we refer the reader to, for example, [4].

Definition 3. An equivalence relation $R \subseteq S \times S$ is a probabilistic bisimulation if for all $(s, t) \in R$, $\ell(s) = \ell(t)$ and there exists $\omega \in \Omega(\tau(s), \tau(t))$ such that $\text{support}(\omega) \subseteq R$, where $\text{support}(\omega) = \{ (u, v) \in S \times S \mid \omega(u, v) > 0 \}$. Probabilistic bisimilarity, denoted $\sim$, is the largest probabilistic bisimulation.

3 Probabilistic Bisimilarity Distances

The probabilistic bisimilarity pseudometric of Desharnais et al. [12] maps each pair of states of a labelled Markov chain to a distance, which an element of the unit interval $[0, 1]$. Hence, the pseudometric is a function from $S \times S$ to $[0, 1]$, that is, an element of $[0, 1]^{S \times S}$. Such a function is a pseudometric if it satisfies the following three properties: for all $s, t, u \in S,$

\footnote{In the literature, different terminology is used. For example, in [1] these are called matchings.}
Theorem 9 holds also for total policies that map to arbitrary elements of the convex polytope $\Omega(\tau(s), \tau(t))$, rather than just its vertices (see [1, Remark 13]). For each $T \in \mathcal{T}$, the pair $\langle S \times S, T \rangle$ can be viewed as a Markov chain, by extending $T$ such that

$$T(s,t)(u,v) = \begin{cases} 1 & \text{if } (u,v) = (s,t) \\ 0 & \text{otherwise} \end{cases}$$

for all $(s,t) \in (S \times S) \setminus S^2_\ell$.

Footnote 2: In the literature, different terminology is used. For example, in [1] these are called couplings.
Definition 8. Let $T \in \mathcal{T}$. The function $\Gamma^T : [0, 1]^{S \times S} \to [0, 1]^{S \times S}$ is defined by

$$
\Gamma^T(d)(s, t) = \begin{cases} 
0 & \text{if } s \sim t \\
1 & \text{if } \ell(s) \neq \ell(t) \\
\sum_{u, v \in S} T(s, t)(u, v) d(u, v) & \text{otherwise}
\end{cases}
$$

Since $[0, 1]^{S \times S}$ is a complete lattice and $\Gamma^T$ is a monotone function [1], we can conclude from the Knaster-Tarski fixed point theorem that $\Gamma^T$ has a least fixed point. We denote this fixed point by $\gamma^T$. Note that $\gamma^T(s, t)$ captures the probability of reaching any $(u, v)$ with $\ell(u) \neq \ell(v)$ from $(s, t)$ in the Markov chain $\langle S \times S, T \rangle$ (see, for example, [3, Section 10.1.1] for a discussion of reachability probabilities). The probabilistic bisimilarity pseudometric $\delta$ can be characterized as follows.

Theorem 9. $\delta = \min_{T \in \mathcal{T}} \gamma^T$.

The Ellipsoid Method

As shown by Chen, Van Breugel and Worrell [8], the probabilistic bisimilarity distances can be computed in polynomial time. In particular, they show that the distances can be obtained by solving a linear programming problem by means of Khachiyan’s ellipsoid method [17]. It is well known that the method is simpler when the feasible region is bounded and full-dimensional (see, for example, [26, Chapter 13]). Below, we will show that the feasible region is bounded and full-dimensional in the setting of Chen et al. This feasible region is defined by

1. $d(s, t) \geq 0$ 
2. $d(s, t) \leq 1$ 
3. $d(s, t) \leq \sum_{(u, v) \in S_2^2} d(u, v) \pi(u, v) + \sum_{(u, v) \in S_2^2} \pi(u, v)$

for all $(s, t) \in S_2^2$ and $\pi \in V(\Omega(\tau(s), \tau(t)))$. Note that we need to decide probabilistic bisimilarity in order to define $S_2^2$ and, hence, the feasible region.

We restrict our attention to the case that the set $S_2^2$ is nonempty. Otherwise, there are no distances to compute. Obviously, the feasible region is bounded by $[0, 1]^{S_2^2}$. A feasible region is full-dimensional if and only if there are no implicit inequalities (see, for example, [26, page 101]). An inequality defining the feasible region is implicit if the inequality is actually an equality for each point of the feasible region (see, for example, [26, page 99] for a formal definition).

Proposition 10. 1. For all $(s, t) \in S_2^2$, there exists $d \in S_2^2 \to \mathbb{R}$ satisfying (1)--(3) such that $d(s, t) > 0$.
2. For all $(s, t) \in S_2^2$, there exists $d \in S_2^2 \to \mathbb{R}$ satisfying (1)--(3) such that $d(s, t) < 1$.
3. For all $(s, t) \in S_2^2$ and $\pi \in V(\Omega(\tau(s), \tau(t)))$, there exists $d \in S_2^2 \to \mathbb{R}$ satisfying (1)--(3) such that

$$d(s, t) < \sum_{(u, v) \in S_2^2} \pi(u, v) d(u, v) + \sum_{(u, v) \in S_2^2} \pi(u, v)$$

Theorem 11. The feasible region defined by (1)--(3) is full-dimensional.
Recall that we need to precompute probabilistic bisimilarity to define the feasible region. If, instead, we were not to do so, then we would have to replace (3) with
\[
d(s, t) \leq \sum_{(u, v) \in S^2 \setminus S^2_1} d(u, v) \pi(u, v) + \sum_{(u, v) \in S^2_1} \pi(u, v)
\]
Now consider the labelled Markov chain consisting of a single state \(s\). In that case the above inequality amounts to \(d(s, s) \leq d(s, s)\) which is implicit and, hence, the feasible region is not full-dimensional.

5 Policy Iteration

In this section we present our modification of the basic algorithm of Bacci et al. [1]. Our optimization will be discussed in Section 6. Before we can present our algorithm, we need one more ingredient.

▶ Definition 12. The function \(\Lambda : [0, 1]^{S \times S} \rightarrow [0, 1]^{S \times S}\) is defined by
\[
\Lambda(d)(s, t) = \begin{cases} 
0 & \text{if } s \sim t \\
\Delta(d)(s, t) & \text{otherwise}
\end{cases}
\]
The following result is proved in [8, Proposition 17]. This result will be instrumental in several proofs later in this paper.

▶ Lemma 13. \(\delta\) is the unique fixed point of \(\Lambda\).

Now that we have all the ingredients, we can present our modification of the basic algorithm by Bacci et al. to compute \(\delta\). This modification was first presented in [28].

1. for each \((s, t) \in S^2\)
2. \(T(s, t) \leftarrow \) an element of \(V(\Omega(\tau(s), \tau(t)))\)
3. while \(\exists (s, t) \in S^2 : \Lambda(\gamma^T)(s, t) < \gamma^T(s, t)\)
4. \(T(s, t) \leftarrow \arg \min_{\omega \in V(\Omega(\tau(s), \tau(t)))} \sum_{u, v \in S} \omega(u, v) \gamma^T(u, v)\)

The only difference with the algorithm of Bacci et al. is that we use \(\Lambda\) instead of \(\Delta\) in line 3. In line 2, for each \((s, t) \in S^2\), a vertex of \(\Omega(\tau(s), \tau(t))\) can be easily obtained in polynomial time by using, for example, Hitchcock’s north west corner rule [14]. As we already mentioned, the fixed point \(\gamma^T\), used in line 3 and 4, captures reachability probabilities and it is well known that these can be computed in polynomial time by solving a system of linear equations (see, for example, [3, Section 10.1.1]). Let \((s, t) \in S^2\). To compute \(\Lambda(\gamma^T)(s, t)\) in line 3 we first have to decide probabilistic bisimilarity. This can be done in polynomial time as has been shown by Baier in [2]. Computing \(\Lambda(\gamma^T)(s, t)\) boils down to solving a minimum cost network flow problem, where \(\omega\) represents the flow and \(\gamma^T\) captures the cost. This problem can be solved in polynomial time using, for example, Orlin’s network simplex algorithm [23]. This algorithm not only computes the minimum cost but also a vertex \(\omega\) of \(\Omega(\tau(s), \tau(t))\) at which that minimum cost is attained, which we use in line 4.

First, we prove the partial correctness of the above algorithm, that is, if the algorithm terminates then it computes the probabilistic bisimilarity distances. Hence, we have to show that at termination \(\gamma^T\) captures \(\delta\). Our proofs here are different from the ones presented in [1, 28] as we will use them as a stepping stone towards proving the partial policy version correct.
Theorem 14. For all $T \in \mathcal{T}$, if $\gamma^T(s,t) \leq \Lambda(\gamma^T)(s,t)$ for all $(s,t) \in S_2^2$, then $\gamma^T = \delta$.

As we already mentioned earlier, for each $(s,t) \in S_2^2$, the set $V(\Omega(\tau(s), \tau(t)))$ is finite. Since we restrict our attention to labelled Markov chains with finitely many states, the set $S_2^2$ is finite as well. Therefore, the set $\mathcal{T}$ is finite. To prove that the loop terminates, we show that $T$ becomes smaller in every iteration.

Definition 15. The order $\prec$ on $\mathcal{T}$ is defined by $T \prec U$ if $\gamma^T \sqsubseteq \gamma^U$.

To relate the value of $T$ at the beginning of the loop with its value at the end of the loop, we introduce the following notation.

Definition 16. Let $T \in \mathcal{T}$, $(s,t) \in S_2^2$ and $\omega \in V(\Omega(\tau(s), \tau(t)))$. The function $T[(s,t)/\omega] : S_2^2 \to \text{Distr}(S \times S)$ is defined by

$$T[(s,t)/\omega](u,v) = \begin{cases} \omega & \text{if } (u,v) = (s,t) \\ T(u,v) & \text{otherwise} \end{cases}$$

Clearly, $T[(s,t)/\omega] \in \mathcal{T}$. Next, we show that $T$ indeed becomes smaller in every iteration of the loop, and as a consequence the loop terminates.

Theorem 17. For all $T \in \mathcal{T}$ and $(s,t) \in S_2^2$, if $\Lambda(\gamma^T)(s,t) < \gamma^T(s,t)$, then $T[(s,t)/\pi] \prec T$, where $\pi = \arg\min_{\omega \in V(\Omega(\tau(s), \tau(t)))} \sum_{u,v \in S} \omega(u,v) \gamma^T(u,v)$.

In the above simple policy iteration algorithm, in each iteration of the loop the policy is adjusted for a single state pair $(s,t)$ which is locally non-optimal, that is, $\Lambda(\gamma^T)(s,t) < \gamma^T(s,t)$. In our general policy iteration algorithm, the policy is updated for all state pairs which are locally non-optimal.

```
for each $(s,t) \in S_2^2$
  $T(s,t) \leftarrow$ an element of $V(\Omega(\tau(s), \tau(t)))$
while $\exists (s,t) \in S_2^2 : \Lambda(\gamma^T)(s,t) < \gamma^T(s,t)$
  $U \leftarrow T$
  for each $(s,t) \in S_2^2$ such that $\Lambda(\gamma^U)(s,t) < \gamma^U(s,t)$
    $T(s,t) \leftarrow \arg\min_{\omega \in V(\Omega(\tau(s), \tau(t)))} \sum_{u,v \in S} \omega(u,v) \gamma^U(u,v)$
```

The proof of partial correctness is the same as for the simple policy iteration algorithm. To prove termination, we slightly generalise Theorem 17.

6 Partial Policy Iteration

As proposed by Bacci et al. in [1], if we are only interested in the probabilistic bisimilarity distances of some of the state pairs, then we may be able to cut down on the number of state pairs for which we need to compute the probabilistic bisimilarity distance. Instead of total policies, we use partial ones. Hence, we generalise the set of total policies $\mathcal{T}$ of Definition 7 as follows. We denote the set of partial functions from $S_2^2$ to $\text{Distr}(S \times S)$ by $S_2^2 \rightarrow \text{Distr}(S \times S)$ and the domain of such a function $P$ by $\text{dom}(P)$.

Definition 18. For a labelled Markov chain $(S, L, \tau, \ell)$, the set $\mathcal{P}$ of partial policies is defined by

$$\mathcal{P} = \{ P \in S_2^2 \rightarrow \text{Distr}(S \times S) \mid \forall (s,t) \in \text{dom}(P) : P(s,t) \in V(\Omega(\tau(s), \tau(t))) \}.$$
Recall that $S^0_1$, $S^1_1$ and $S^2_1$ form a partition of $S \times S$. For a given $P \in \mathcal{P}$, $S^0_1$, $S^1_1$, $S^2_1 \setminus \text{dom}(P)$ and $S^2_1 \cap \text{dom}(P)$ form a partition of $S \times S$ as well. This partition is used to generalise the function $F^\mathcal{T}$ of Definition 8 to the partial setting.

**Definition 19.** Let $P \in \mathcal{P}$. The function $\Theta^P : [0,1]^{S \times S} \rightarrow [0,1]^{S \times S}$ is defined by

$$
\Theta^P(d)(s,t) = \begin{cases} 
1 & \text{if } \ell(s) \neq \ell(t) \\
0 & \text{if } s \sim t \lor (\ell(s) = \ell(t) \land s \neq t) \land (s,t) \notin \text{dom}(P) \\
\sum_{u,v \in S} P(s,t)(u,v) d(u,v) & \text{otherwise}
\end{cases}
$$

Since $[0,1]^{S \times S}$ is a complete lattice and $\Theta^P$ is a monotone function, we can conclude from the Knaster-Tarski fixed point theorem that $\Theta^P$ has a least fixed point. We denote this fixed point by $\theta^P$.

The set $Q \subseteq S^2_1$ contains those pairs of states for which we want to compute their distances. Recall that for pairs of states in $(S \times S) \setminus S^2_1 = S^0_1 \cup S^1_1$ we can obtain their distances by deciding probabilistic bisimilarity and comparing their labels.

1. $P \leftarrow$ the partial function with empty domain
2. for each $(s,t) \in Q$
3. $P(s,t) \leftarrow$ an element of $V(\Omega(\tau(s),\tau(t)))$
4. expand($P$, $s$, $t$)
5. while $\exists (u,v) \in \text{dom}(P) : \Lambda(\theta^P)(s,t) < \theta^P(s,t)$
6. $P(s,t) \leftarrow \arg\min_{u \in V(\Omega(\tau(s),\tau(t)))} \sum_{u,v \in S} \omega(u,v) \theta^P(u,v)$
7. expand($P$, $s$, $t$)

Let $P \in \mathcal{P}$ and $(s,t) \in S^2_1$. The recursive function $\text{expand}(P,s,t)$ is defined as follows.

8. while $\exists (u,v) \in \text{support}(P(s,t)) \cap S^2_1 : (u,v) \notin \text{dom}(P)$
9. $P(u,v) \leftarrow$ an element of $V(\Omega(\tau(u),\tau(v)))$
10. expand($P$, $u$, $v$)

To prove properties of this recursive function, we introduce the following predicate.

**Definition 20.** Let $P \in \mathcal{P}$ and $X \subseteq S^2_1$. The predicate $F(P,X)$ is defined by

$$
F(P,X) = \forall(s,t) \in \text{dom}(P) \setminus X : \text{support}(P(s,t)) \cap S^2_1 \subseteq \text{dom}(P).
$$

Roughly, this predicate $F(P,X)$ captures that $P$ is fully defined when we exclude $X$ from its domain. Let $P \in \mathcal{P}$ and $(s,t) \in S^2_1$. Next, we prove that for $\text{expand}(P,s,t)$ the precondition $F(P,X)$ implies the postcondition $F(P,X \setminus \{(s,t)\})$. First, we observe that $F(P,X)$ is a loop invariant. At the start of line 10, we have that $F(P,X \cup \{(u,v)\})$. Hence, at the end of line 10 we have $F(P,X)$. To conclude that the loop terminates, we observe that the finite set $\text{dom}(P) \setminus \text{support}(P(s,t))$ becomes smaller in every iteration. Note that expand does not give rise to infinite recursion since for each recursive call the finite set $S^2_1 \setminus \text{dom}(P)$ becomes smaller. At the end of the loop we have $F(P,X)$ and $(u,v) \in \text{dom}(P)$ for all $(u,v) \in \text{support}(P(s,t)) \cap S^2_1$, that is, $\text{support}(P(s,t)) \cap S^2_1 \subseteq \text{dom}(P)$. Therefore, $F(P,X \setminus \{(s,t)\})$.

The proof of partial correctness of this simple partial policy iteration algorithm is similar to the partial correctness proof provided in Section 5. If the above algorithm terminates, then we have that

$$
F(P,\emptyset) \land Q \subseteq \text{dom}(P) \land \forall(s,t) \in \text{dom}(P) : \theta^P(s,t) \leq \Lambda(\theta^P)(s,t)
$$
at termination. As we will show next, from the above we can conclude that \( \theta^P \) and \( \delta \) coincide on \( Q \) and, hence, \( \theta^P \) contains the probabilistic bisimilarity distances of \( Q \).

**Theorem 21.** For all \( P \in \mathcal{P} \), if \( F(P, \emptyset) \) and \( \theta^P(s, t) \leq \Lambda(\theta^P)(s, t) \) for all \( (s, t) \in \text{dom}(P) \), then \( \theta^P(s, t) = \delta(s, t) \) for all \( (s, t) \in \text{dom}(P) \).

It remains to prove that the simple partial policy iteration algorithm terminates. As we already discussed above, the recursive function \( \text{expand} \) terminates. Hence, we are left to show that the loop of line 5–7 terminates as well. We prove this by showing that in each iteration of the loop \( (S^2 \setminus \text{dom}(P), P) \) becomes smaller. These pairs are ordered lexicographically, with the first component ordered by \( \subset \) and the second component ordered by \( \prec \), as introduced in Definition 15. Assume that \( P \) is updated for \( (s, t) \) in line 6 of the current iteration of the loop. We distinguish two cases. If \( (u, v) \notin \text{dom}(P) \) for some \( (u, v) \in \text{support}(P(s, t)) \cap S^2 \), then \( \text{expand}(P, s, t) \) in line 7 will assign a value to \( P(u, v) \) in line 9 of the expand function. As a consequence, \( \text{dom}(P) \) becomes bigger and, hence, \( S^2 \setminus \text{dom}(P) \) becomes smaller, that is, the first component becomes smaller. Note that in this case \( P \) may not become smaller as \( \theta^P(u, v) \) was zero and may have become positive. Otherwise, \( \text{support}(P(s, t)) \cap S^2 \subseteq \text{dom}(P) \). In that case, the expand function does not perform any assignments to \( P \) and, therefore, \( \text{dom}(P) \) stays the same. Thus, the first component stays the same. Furthermore, the iteration changes the partial policy from \( P \) to \( P[(s, t)/\pi] \) (cf. Definition 16). As we show next, in this case the second component, that is, the partial policy, becomes smaller.

**Theorem 22.** For all \( P \in \mathcal{P} \) and \( (s, t) \in \text{dom}(P) \), if \( \Lambda(\theta^P)(s, t) < \theta^P(s, t) \), then \( P[(s, t)/\pi] < P \), where \( \pi = \arg \min_{\omega \in V(\Omega(\tau(s), \tau(t)))} \sum_{u, v \in S} \omega(u, v) \theta^P(u, v) \).

The algorithm of Bacci et al. differs in three major ways from our simple partial policy iteration algorithm. First of all, as we already mentioned in Section 5, they use \( \Lambda \) instead of \( \Delta \) in line 5. In [28, Theorem 8] we give an example different from the one presented below which shows that \( \Lambda \) is essential for computing the distances correctly. Secondly, in line 5 they consider only those state pairs \( (s, t) \) that are reachable from state pairs in \( Q \) in the Markov chain \( (S \times S, P) \), instead of those in \( \text{dom}(P) \). But, as we show below, as a result they do not always correctly compute the distances. Thirdly, they assign one to \( \theta^P(s, t) \) when \( (s, t) \notin \text{dom}(P) \) whereas we assign zero. An example similar to the one presented below can be used to show that this also gives to computing the distances incorrectly.

We conclude this section with an example that shows that Bacci et al. do not always consider partial policies that are defined for sufficiently many state pairs. Consider the labelled Markov chain presented in the introduction. Assume that we are only interested in the probabilistic bisimilarity distance between the states \( s \) and \( t \). That is, \( Q = \{(s, t)\} \). After executing line 1–4 of the simple partial policy iteration algorithm, we may end up with the partial policy \( P \) defined by

\[
\begin{align*}
P(s, t) &= \frac{1}{2}\text{Dir}(u, v) + \frac{1}{2}\text{Dir}(a, d) + \frac{1}{2}\text{Dir}(b, c) \quad P(a, d) = \frac{1}{4}\text{Dir}(a_1, d_1) + \frac{1}{4}\text{Dir}(a_2, d_1) \\
P(b, c) &= \frac{1}{2}\text{Dir}(b_1, c_2) + \frac{1}{2}\text{Dir}(b_2, c_1)
\end{align*}
\]

where the Dirac distribution \( \text{Dir}(u, v) \) is defined by

\[
\text{Dir}(u, v)(x, y) = \begin{cases} 1 & \text{if } (x, y) = (u, v) \\ 0 & \text{otherwise} \end{cases}
\]

At this point, we have \( \theta^P(s, t) = 1, \theta^P(a, d) = 1 \) and \( \theta^P(b, c) = 1 \). Note that \( (s, t) \) is not locally optimal, that is, \( \Lambda(\theta^P)(s, t) < \theta^P(s, t) \). We update \( P \) by setting \( P(s, t) = \frac{1}{2}\text{Dir}(a, c) + \frac{1}{2}\text{Dir}(b, c) \)
\[ P(a, c) = \frac{1}{2} \text{Dir}(a_1, c_1) + \frac{1}{2} \text{Dir}(a_2, c_2) \quad P(b, d) = \frac{1}{2} \text{Dir}(b_1, d_1) + \frac{1}{2} \text{Dir}(b_2, d_2) \]

At this point, we have \( \theta P(s, t) = \frac{3}{4} \), \( \theta P(a, c) = 1 \) and \( \theta P(b, d) = \frac{1}{2} \). Since in their algorithm, Bacci et al. only check local optimality for all state pairs reachable from \((s, t)\) in the Markov chain \( (S \times S, P) \), that is, for \((s, t)\), \((a, c)\) and \((b, d)\), and all three are locally optimal, their algorithm terminates at this point. Our algorithm checks for local optimality for all state pairs in \( \text{dom}(P) \). Since neither \((a, d)\) nor \((b, c)\) are locally optimal, our algorithm continues. We update \( P \) by setting

\[ P(a, d) = \frac{1}{2} \text{Dir}(a_1, d_1) + \frac{1}{2} \text{Dir}(a_2, d_2) \quad P(b, c) = \frac{1}{2} \text{Dir}(b_1, c_1) + \frac{1}{2} \text{Dir}(b_2, c_2) \]

At this point, we have \( \theta P(s, t) = \frac{3}{4} \), \( \theta P(a, d) = \frac{1}{2} \) and \( \theta P(b, c) = \frac{1}{2} \). Since \((s, t)\) is not locally optimal any more, we update \( P \) by setting \( P(s, t) = \frac{1}{2} \text{Dir}(a_d) + \frac{1}{2} \text{Dir}(b_c) \). This results in \( \theta P(s, t) = \frac{1}{2} \) which is the probabilistic bisimilarity distance of \((s, t)\).

Also the general policy iteration algorithm, which we presented at the end of Section 5, can be generalised to use partial policies instead of total ones.

### 7 An Exponential Lower Bound

Below, we analyze the worst case running time of the simple partial policy iteration algorithm. We show that it is at least exponential in the number of states of the labelled Markov chain.

**Definition 23.** For \( n \in \mathbb{N} \), the labelled Markov chain \( \mathcal{M}_n \) is defined as follows by induction on \( n \). The labelled Markov chain \( \mathcal{M}_0 \) is defined as

If \( n > 0 \) then the labelled Markov chain \( \mathcal{M}_n \) is defined as

where the two dashed triangles together represent the labelled Markov chain \( \mathcal{M}_{n-1} \).
The above labelled Markov chain $\mathcal{M}_n$ has $4n + 10$ states and $7n + 14$ transitions. Next, we show that it may take at least $2^{n+1} - 1$ iterations of the simple partial policy iteration algorithm to compute the distance of $s_n$ and $t_n$ in $\mathcal{M}_n$.

The partial policy iteration algorithm contains some non-determinism. In particular, in line 3 and 9, an element of $V(\Omega(\tau(s), \tau(t)))$ and $V(\Omega(\tau(u), \tau(v)))$ is chosen. Furthermore, in line 5 a state pair $(s, t) \in \text{dom}(P)$ with $\Lambda(\theta^P)(s, t) < \theta^P(s, t)$ is selected. Note that, for all $1 \leq i \leq n$,

$$V(\Omega(\tau(s_i), \tau(t_i))) = \{ \frac{1}{2} \text{Dir}(s_{i-1}, t_{i-1}) + \frac{1}{2} \text{Dir}(v_{i-1}, u_{i-1}), \frac{1}{2} \text{Dir}(s_{i-1}, u_{i-1}) + \frac{1}{2} \text{Dir}(v_{i-1}, t_{i-1}) \}.$$

Also,

$$V(\Omega(\tau(s_0), \tau(t_0))) = \{ \frac{1}{2} \text{Dir}(s_{-1}, u_{-1}) + \frac{1}{2} \text{Dir}(v_{-1}, t_{-1}), \frac{1}{2} \text{Dir}(s_{-1}, t_{-1}) + \frac{1}{2} \text{Dir}(v_{-1}, u_{-1}) \}.$$

Furthermore, for all $1 \leq i < n$,

$$V(\Omega(\tau(s_i), \tau(u_i))) = \{ \frac{1}{2} \text{Dir}(v_{i-1}, u_{i-1}) + \frac{1}{2} \text{Dir}(s_{i-1}, t_{i-1}), \frac{1}{2} \text{Dir}(s_{i-1}, u_{i-1}) + \frac{1}{2} \text{Dir}(v_{i-1}, t_{i-1}) \}.$$

To realize the exponential lower bound, in line 3 and 9 we choose the first element of the above sets and in line 5 we select the $(s_i, t_i)$ with maximal index $i$.

**Theorem 24.** For each $n \in \mathbb{N}$, there exists a labelled Markov chain of size $O(n)$ and a singleton set $Q$ such that simple partial policy iteration takes $\Omega(2^n)$ iterations to compute the distances for the state pair in $Q$.

### 8 Experimental Results

Next we present results from experiments comparing the performance of six different algorithms: the algorithm described in the introduction by Van Breugel et al. [5] based on the first order theory over the reals, the algorithm by Chen et al. [8] based on the ellipsoid method, our simple policy iteration algorithm, our general policy iteration algorithm, our simple partial policy iteration algorithm and our general partial policy iteration algorithm. We implemented all the algorithms in Java. In our implementations we do not use arbitrary precision arithmetic. These implementations were run on a number of labelled Markov chains. These labelled Markov chains model well-known randomized algorithms and were obtained from examples of probabilistic model checkers such as PRISM [20] and jpf-probabilistic [31].

For each labelled Markov chain, we executed the code ten times. The first few executions were discarded to account for the “warm-up” time that the Java virtual machine needs to perform just-in-time compilation and optimization. For the remaining runs the average running time and the standard deviation were computed for each labelled Markov chain.

Since the distance function is symmetric and the distance from a state to itself is zero, we only need to compute the distance of $\frac{n^2 - n}{2}$ state pairs for a labelled Markov chain with $n$ states. Recall that the policy iteration algorithms precompute distances for state pairs that are probabilistic bisimilar or have different labels. Hence, only state pairs with the same label that are not probabilistic bisimilar are considered. In general, the partial policy iteration algorithms compute the distances for even fewer state pairs. For our experiments, we report for each policy iteration algorithm how many state pairs it considers and how many iterations it takes.
The first example we consider is a version of quicksort in which the pivot is chosen randomly. An implementation of this algorithm is part of jpf-probabilistic and this tool can be used to obtain the corresponding labelled Markov chain. The size of the labelled Markov chain grows exponentially in the size of the input, which is the list to be sorted. For example, lists of size 4, 5 and 6 give rise to labelled Markov chains with 10, 28 and 82 states, respectively.

The first order theory over the reals algorithm can only handle labelled Markov chains with a handful of states. We ran the algorithm on the labelled Markov chain with 10 states. It did not terminate within three days. The ellipsoid method takes on average 73 seconds. For the partial algorithms we compute the distance for a single pair of states. Of the 45 state pairs, the policy iteration algorithms consider 8 state pairs and the partial policy iteration algorithms only 3 state pairs. All policy iteration algorithms take less than 45 milliseconds. That makes the ellipsoid method three orders of magnitude slower than our policy iteration algorithms for this small example.

We did not run the first order theory over the reals algorithm on the randomized quicksort example with 28 states. The ellipsoid method takes more than 43 hours, making it five orders of magnitude slower than the policy iteration algorithms, which take less than a dozen seconds. For the example with 82 states, of the 3,321 state pairs the policy iteration algorithms consider 870 states and the partial policy iteration algorithms only 13 states.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>List size</th>
<th>Running time</th>
<th>Standard deviation</th>
<th>State pairs</th>
<th>Iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple</td>
<td>4</td>
<td>42.75 ms</td>
<td>5.96 ms</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>General</td>
<td>4</td>
<td>27.93 ms</td>
<td>4.48 ms</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>Simple partial</td>
<td>4</td>
<td>12.23 ms</td>
<td>2.84 ms</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>General partial</td>
<td>4</td>
<td>8.84 ms</td>
<td>2.25 ms</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Simple</td>
<td>5</td>
<td>12.00 s</td>
<td>19.38 ms</td>
<td>99</td>
<td>13</td>
</tr>
<tr>
<td>General</td>
<td>5</td>
<td>4.58 s</td>
<td>20.39 ms</td>
<td>99</td>
<td>13</td>
</tr>
<tr>
<td>Simple partial</td>
<td>5</td>
<td>51.58 ms</td>
<td>2.51 ms</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>General partial</td>
<td>5</td>
<td>81.62 ms</td>
<td>5.67 ms</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Simple</td>
<td>6</td>
<td>15.61 h</td>
<td>4.00 m</td>
<td>870</td>
<td>159</td>
</tr>
<tr>
<td>General</td>
<td>6</td>
<td>47.30 m</td>
<td>6.03 s</td>
<td>870</td>
<td>154</td>
</tr>
<tr>
<td>Simple partial</td>
<td>6</td>
<td>24.48 s</td>
<td>162.91 ms</td>
<td>13</td>
<td>3</td>
</tr>
<tr>
<td>General partial</td>
<td>6</td>
<td>53.63 s</td>
<td>203.58 ms</td>
<td>13</td>
<td>4</td>
</tr>
</tbody>
</table>

In [19], Knuth and Yao show how to model a die by means of a fair coin. An implementation of their algorithm is part of PRISM. The resulting labelled Markov chain has 13 states.
In the next experiment, we model two dice, one using only a fair coin and the other one using a biased coin with probability 0.6 for heads and 0.4 for tails. The goal is to compute the probabilistic bisimilarity distance between the two dice. The resulting labelled Markov chain has 20 states.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Running time</th>
<th>Standard deviation</th>
<th>State pairs</th>
<th>Iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple</td>
<td>10.99 s</td>
<td>69.67 ms</td>
<td>91</td>
<td>51</td>
</tr>
<tr>
<td>General</td>
<td>1.51 s</td>
<td>20.45 ms</td>
<td>91</td>
<td>51</td>
</tr>
<tr>
<td>Simple partial</td>
<td>1.28 s</td>
<td>12.00 ms</td>
<td>21</td>
<td>17</td>
</tr>
<tr>
<td>General partial</td>
<td>0.55 s</td>
<td>19.83 ms</td>
<td>21</td>
<td>18</td>
</tr>
</tbody>
</table>

It can be seen from the above experiments that the simple policy iteration algorithm is often slower than the general policy iteration algorithm. Moreover, if we are only interested in the probabilistic bisimilarity distances between a few states, the partial policy iteration algorithms are much more efficient as only part of the labelled Markov chain is considered and only the distances of related pairs of states are computed.

## 9 Conclusion

Although behavioural equivalences like probabilistic bisimilarity are not robust, they still play a pivotal role when computing their quantitative generalisations. For example, as we have seen in Section 4, the feasible region is defined in terms of probabilistic bisimilarity. The policies used in the policy iteration algorithms are only defined for states that are not probabilistic bisimilar. Hence, in both cases one first has to decide probabilistic bisimilarity. Deciding probabilistic bisimilarity takes only a fraction of the time to compute the probabilistic bisimilarity distances. For all the examples discussed in Section 8, deciding probabilistic bisimilarity takes less than 50 milliseconds.

The probabilistic bisimilarity pseudometric of Desharnais et al. also has discounted variants (see [12] for details). Bacci et al. consider the undiscounted version, as we do in this paper, as well as the discounted variants in [1]. All the results presented in this paper carry over to the discounted setting.

In [28] we prove an exponential lower bound for the simple policy iteration algorithm. In this paper, we present an exponential lower bound for our simple partial policy iteration algorithm. It is still open whether there exists an exponential lower bound for our general policy iteration algorithm.

**Acknowledgements.** The authors are thankful to all the referees of this paper for their constructive feedback.
References


27:16 Probabilistic Bisimilarity Distances


On Decidability of Concurrent Kleene Algebra∗†

Paul Brunet1, Damien Pous2, and Georg Struth3

1 Univ. Lyon, CNRS, ENS de Lyon, UCB Lyon 1, LIP, France
2 Univ. Lyon, CNRS, ENS de Lyon, UCB Lyon 1, LIP, France
3 Department of Computer Science, The University of Sheffield, UK

Abstract

Concurrent Kleene algebras support equational reasoning about computing systems with concurrent behaviours. Their natural semantics is given by series(-parallel) rational pomset languages, a standard true concurrency semantics, which is often associated with processes of Petri nets. We use constructions on Petri nets to provide two decision procedures for such pomset languages motivated by the equational and the refinement theory of concurrent Kleene algebra. The contribution to the first problem lies in a much simpler algorithm and an \textit{ExpSpace} complexity bound. Decidability of the second, more interesting problem is new and, in fact, \textit{ExpSpace}-complete.

1998 ACM Subject Classification F.3.1 Specifying, verifying, and reasoning about programs

Keywords and phrases Concurrent Kleene algebra, series-parallel pomsets, Petri nets

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2017.28

1 Introduction

Kleene algebras axiomatise the equational theory of rational expressions. Their canonical models are rational languages and their equational theories correspond to rational expression equivalence [12, 11, 1, 2]. Deciding identities in Kleene algebras is therefore \textit{PSpace}-complete [17] by standard automata constructions. Variants of Kleene algebras provide simple algebraic semantics for while-programs, and, in particular, decision procedures for these.

Pomset languages [6], on the other hand, are a widely studied model of true concurrency in which words are generalised from linear orders to partial ones. Recent applications can be found, for instance, in weak memory model verification [9]. Algebras for pomsets have been proposed first by Gischer [5] and more recently, as concurrent Kleene algebra (CKA), by Hoare et al. [8], with the aim of extending the pleasant properties of Kleene algebras into concurrency. Yet much less is known about their structure.

Formally, CKAs are structures \((K, +, \cdot, ∥, ⋆, 0, 1)\) that consist of a Kleene algebra \((K, +, ⋆, 0, 1)\) and a commutative Kleene algebra \((K, +, ∥, 0, 1)\), and satisfy the weak interchange law defined below. Commutative Kleene algebras axiomatise rational commutative expression equivalence, which is decidable [4] and \textit{coNExp}-complete [7]. In applications of CKA, the elements of \(K\) are typically actions of a system: The operation \(+\) models nondeterministic choices, \(\cdot\) and \(∥\) sequential and parallel compositions, 1 the ineffective action, and 0 the abortive one. The sequential star \(⋆\) models the finite sequential iteration of actions.

∗ An extended version of this abstract, including proofs, is available on HAL [3].
† This work was supported by the European Research Council (ERC) under the Horizon 2020 programme (CoVeCe, grant agreement No 678157) and the LABEX MILYON (ANR-10-LABX-0070) of Université de Lyon, within the program “Investissements d’Avenir” (ANR-11-IDEX-0007).
in terms of a least fixpoint, the parallel star \((\ast)\) their finite parallel iteration. It can be interpreted as the unbounded spawning of parallel processes.

Closed terms in the language of Kleene algebra correspond to rational expressions; their interpretation as word languages is standard. The extension to parallelism, hence to action-labelled partial orders and pomsets, is best explained by example. The expression \((a \cdot b) \parallel c\), for instance, is represented by the first of the following pomsets.

\[
\begin{array}{c}
\text{a} \\
\text{c}
\end{array}
\quad
\begin{array}{c}
\text{a} \\
\text{d}
\end{array}
\quad
\begin{array}{c}
\text{b} \\
\text{d}
\end{array}
\quad
\begin{array}{c}
\text{d}
\end{array}
\]

Execution time—the order of the poset—is indicated by lines proceeding from left to right in this implicitly directed graph. Sequential composition thus orders actions, whereas parallel composition leaves them unordered. By analogy to word languages, expressions involving + or the stars require interpretations by sets of pomsets, that is, pomset languages. The expression \((a \cdot b) \parallel (c + (d \cdot d))\), for instance, denotes the language formed by the first two of the pomsets above. The third of the above pomsets is denoted by \((a \parallel d) \cdot (b \parallel d)\). It is obviously “more sequential” than the poset to its left, which is denoted by \((a \cdot b) \parallel (d \cdot d)\). A corresponding refinement order, which compares degrees of sequentiality, has been defined on pomsets (as the smoother-than relation) by Grabowski [6]. It is isomorphic to the inclusion order on refinement-closed pomset languages and induces a (refinement) order on CKA expressions. Gischer [5] has shown that this order is characterised precisely by the inequality \((a \parallel c) \cdot (b \parallel d) \leq (a \cdot b) \parallel (c \cdot d)\) on CKA expressions (without the stars). This weak interchange law is also one of the standard CKA axioms.

Pomset languages are typically infinite when expressions contain stars. In addition, the width of individual pomsets can be unbounded when parallel stars occur; this star is therefore often omitted [15]. Furthermore, CKA expressions generate subclasses of pomset languages. Those generated by expressions over the full CKA signature are called series-parallel-rational (spr-languages), those generated by using a signature without the parallel star are called series-rational (sr-languages). Expressions are named accordingly. All pomsets occurring in spr- or sr-languages, which are built inductively from singleton pomsets by sequential and parallel compositions, are series-parallel or, equivalently, free of N-shape subpomsets [19, 6].

Equivalence of spr-expressions, as induced by spr-language identity, is decidable and can be axiomatised by any set of axioms for Kleene algebras plus those for commutative Kleene algebras [13], but a reasonable upper complexity bound has not been established. In the context of CKA with the interchange axiom, completeness or decidability of the refinement of spr-expressions, or even sr-expressions, as induced by inclusion of refinement-closed spr- or sr-languages, remains open. These questions are of obvious interest for comparing concurrent systems with respect to their degree of sequentiality or linearisability.

Our first contribution consists in a simple new algorithm and a first complexity bound for sr-expression equivalence. First, using a construction similar to Thompson’s [18] and Grabowski’s [6], we show that every sr-language is the pomset trace language of a safe labelled Petri net. Using a result by Jategaonkar and Meyer on pomset languages of Petri nets [10], it then follows that sr-expression equivalence is in \(\text{ExpSpace}\) (Theorem 5).

Our second, more interesting contribution is a proof that sr-expression refinement is \(\text{ExpSpace}\)-complete (Theorem 26). Note that sr-expression equivalence is sr-expression refinement in both directions. This result requires comparing runs in Petri nets up-to Grabowski’s refinement order, using the freedom provided by this formalism to reorder transitions, and a schedule for constructing a comparison function in a canonical way. Preservation of sequentiality or causality in this construction is somewhat intricate: it
requires tracking the history and relationships between \textit{loci} (Section 5.2 and 5.3). The Petri net approach seems natural once more due to the correspondence between nets and pomset languages, and our previous construction. Hardness of \textit{sr}-expression refinement follows from a reduction from the equivalence problem for regular expressions with a shuffle operation \cite{16}, using results by Grabowski that relate pomset and shuffle languages.

2 Preliminary definitions

2.1 Pomsets

We fix a finite alphabet \(\Sigma\). A \textit{labelled poset} is a triple \(\langle X, \leq, \lambda \rangle\) where \(X\) is a finite carrier set, \(\leq\) is a partial order on \(X\) and the map \(\lambda : X \to \Sigma\) labels every element in \(X\) with a letter in \(\Sigma\). A \textit{(labelled poset) morphism} is a function between labelled posets that preserves the order and the labels. A \textit{pomset} is an isomorphism class of labelled posets; it is a labelled poset up-to bijective renaming of the elements in \(X\). We represent pomsets as graphs that are implicitly directed from left to right. The vertices, which are the elements of the pomset, are labelled by \(\lambda\); those edges that can be deduced by transitivity and reflexivity are omitted.

We define the following pomsets and operations on pomsets:

- The \textit{empty pomset}, denoted by \(P_0\), is defined as \(\langle \emptyset, \emptyset, [] \rangle\) (\([\ ]\) denoting the empty function);
- for \(a \in \Sigma\), the \textit{singleton pomset} \(P_a\) is \(\langle \{\bullet\}, \{\bullet, \bullet\}, \bullet \mapsto a \rangle\);
- for pomsets \(P_1 = \langle X_1, \leq_1, \lambda_1 \rangle\) and \(P_2 = \langle X_2, \leq_2, \lambda_2 \rangle\) with \(X_1 \cap X_2 = \emptyset\); the \textit{parallel product} of \(P_1\) and \(P_2\) is the pomset obtained by putting them side by side:

\[
P_1 \parallel P_2 \triangleq \langle X_1 \cup X_2, \leq_1 \cup \leq_2, \lambda_1 \cup \lambda_2 \rangle ;
\]

- the \textit{sequential product} of \(P_1\) and \(P_2\) is the pomset obtained by further declaring all elements of \(P_1\) as smaller than those of \(P_2\):

\[
P_1 ; P_2 \triangleq \langle X_1 \cup X_2, \leq_1 \cup \leq_2 \cup X_1 \times X_2, \lambda_1 \cup \lambda_2 \rangle .
\]

Pomset \(P_1\) \textit{refines} \(P_2\), written \(P_1 \sqsubseteq P_2\), if there exists a bijective morphism \(\varphi : X_2 \to X_1\). By definition, therefore,

- \(\forall x \in X_2, \lambda_1(\varphi(x)) = \lambda_2(x)\), i.e., the bijection preserves labels; and
- \(\forall x, y \in X_2, x \leq_2 y \Rightarrow \varphi(x) \leq_1 \varphi(y)\), i.e., the morphism preserves edges in \(P_2\).

The relation \(\sqsubseteq\) is a partial order on pomsets. We write \(\sqsubseteq S\) for the downward closure of a set \(S\) of pomsets with respect to it: \(\sqsubseteq S \triangleq \{P \mid \exists Q : P \subseteq Q, Q \in S\}\). We then extend the refinement order to a preorder on sets of pomsets: \(S \sqsubseteq S' \triangleq S \subseteq \sqsubseteq S'\). (This definition is equivalent to \(S \sqsubseteq S' \triangleq S \sqsubseteq \sqsubseteq S'\).

2.2 Expressions and pomset languages

A \textit{series-rational expression}, or more briefly expression, is a term derived from the following syntax. The set of expressions over the alphabet \(\Sigma\) is written \(\text{Rat}^{|\Sigma|}\).

\[
e, f := e + f \mid e \cdot f \mid e \parallel f \mid e^* \mid 0 \mid 1 \mid a \quad (a \in \Sigma)
\]

The \textit{language} of an expression is the set of pomsets defined inductively as follows:

\[
[1] \triangleq \{P_0\} \quad [e, f] \triangleq \{P : Q \mid P \in [e], Q \in [f]\} \quad [e \parallel f] \triangleq \{P \parallel Q \mid P \in [e], Q \in [f]\}.
\]

\[
[0] \triangleq \emptyset \quad [e + f] \triangleq [e] \cup [f] \quad [e^*] \triangleq \bigcup_{n \in \mathbb{N}} \{P_1, \ldots, P_n \mid \forall i \leq n, P_i \in [e]\} \quad [a] \triangleq \{P_a\}.
\]
A set of pomsets is called \textit{(series-)rational} if it is the language of some expression. It is called \textit{downward-closed rational} if it is the downward-closure of a rational language.

Note that due to the structure of expressions, the pomsets we consider are always \textit{series-parallel}: they are built from trivial pomsets by using sequential and parallel compositions. Valdes et al. proved that this property is equivalent to \textit{N-freeness} \cite{19, 6}: whenever there are four distinct elements \(x, y, z, t\) such that \(x \leq y, z \leq y, \) and \(z \leq t\), then either \(z \leq x, t \leq y, \) or \(x \leq t\).

In the present work we are interested in the following two decision problems.

\textbf{Definition 1.} Given two expressions \(e, f\), the problem \textit{biKA}(\(e, f\)) asks if \([e] \subseteq [f]\).

\textbf{Definition 2.} Given two expressions \(e, f\), the problem \textit{CKA}(\(e, f\)) asks if \([e] \sqsubseteq [f]\).

The first problem, \textit{biKA}(\(e, f\)), asks essentially about equivalence of the sr-expressions \(e\) and \(f\). As outlined in the introduction, axioms for Kleene algebras plus those for commutative Kleene algebras (here in fact commutative idempotent semirings without a parallel star) are complete w.r.t. this equivalence. The second one, \textit{CKA}(\(e, f\)), asks whether \(e\) is a refinement of \(f\), which relates to CKA with the interchange law, yet again without a parallel star. We conjecture that the aforementioned axioms together with weak interchange are complete for this semantics, but this problem remains open, to the best of our knowledge.

\subsection{Labelled safe Petri nets}

We now define \textit{labelled safe Petri nets}—the machines that we use to recognise rational pomset languages. We write \(\mathcal{P}_+(X)\) for the set of non-empty subsets of a set \(X\).

A \textit{labelled Petri net} is a tuple \(\mathcal{N} = (\mathcal{P}, T, p_{\text{in}}, p_{\text{fin}})\) where:

\begin{itemize}
  \item \(\mathcal{P}\) is a finite set of \textit{places};
  \item \(T \subseteq \mathcal{P}_+(\mathcal{P}) \times (\Sigma \cup \{\tau\}) \times \mathcal{P}_+(\mathcal{P})\) is a set of \textit{labelled transitions};
  \item \(p_{\text{in}} \in \mathcal{P}\) is the \textit{initial place};
  \item \(p_{\text{fin}} \in \mathcal{P}\) is the \textit{final place}.
\end{itemize}

If \(t = (P, x, P')\) is a transition, then \(P\) is its \textit{input set}, written \(\bullet t\), \(x\) is its \textit{label}, written \(\ell(t)\), and \(P'\) is its \textit{output set}, written \(t^*\). Transitions labelled with \(\tau\) are called \textit{silent}; the others are called \textit{visible}. Without loss of generality, we may restrict ourselves to Petri nets where all inputs and outputs of visible transitions are singleton sets. An example of such a Petri net is displayed in Figure 1.

A \textit{configuration} is a set of places. A transition \(t\) is \textit{enabled} from a configuration \(C\) if \(\bullet t \subseteq C\). Whenever \(t\) is enabled in \(C\), then firing this transition leads to the configuration.
Figure 2 An accepting run of the Petri net in Figure 1.

Let $R = C_0 \stackrel{t_1, \ldots, t_n}{\rightarrow_R} C_n$ be a run in a Petri net $\mathcal{N}$. We define the immediate causality relation $\rightarrow_R \subseteq [1..n] \times [1..n]$ as

$$i \rightarrow_R j \triangleq i < j \land (\exists p \in \ell^*_i \cap \ell^*_j : \forall k, \; i < k < j \Rightarrow p \notin \ell^*_k).$$

The causality relation $\leq_R$ is the reflexive transitive closure of $\rightarrow_R$. Intuitively, $i \leq_R j$ holds if $t_j$ cannot be fired in a subrun of $R$ without firing $t_i$.

For each run one can define three kinds of traces [10]. For the run from Figure 2, these are shown in Figure 3.

- The graph-trace $G(R)$ of $R$ is the graph $\langle [1..n], \rightarrow_R \rangle$.
- The transition-pomset $R$ is the pomset $T(R) \triangleq \langle [1..n], \leq_R, \lambda_R \rangle$, where $\lambda_R(i) \triangleq \ell(t_i)$.
- The pomset-trace $P(R)$, of $R$ is the restriction of $T(R)$ to the set $\{i \mid \ell(t_i) \in \Sigma\}$ of visible actions.
The pomset language of a Petri net $\mathcal{N}$ is the set $[\mathcal{N}]$ of pomset-traces of accepting runs in $\mathcal{N}$. Moreover, we call a run $R$ is series-parallel if its graph-trace is series-parallel. Note that this is strictly stronger than requiring that its pomset-trace be series-parallel.

The run in Figure 2 is series-parallel.

3 Reading a pomset in a Petri net

This section describes an operational way of reading and recognising pomsets with Petri nets, as one might read and recognise a word with a finite state automaton. It is independent from the rest of the paper, but might provide insight into the algorithm we develop below to compare languages of nets. Indeed, the guiding intuition behind this algorithm will be to read a net in another.

Let $\mathcal{N} = (\mathcal{P}, T, p_{in}, p_{fin})$ be a safe labelled Petri net, $P = \langle X, \leq, \lambda \rangle$ a pomset, and $R = C_0 \overset{t_1}{\rightarrow} C_1 \overset{t_2}{\rightarrow} \cdots C_{n-1} \overset{t_n}{\rightarrow} C_n$ a run in $\mathcal{N}$. A reading of $P$ in $\mathcal{N}$ along $R$ is a sequence $\langle \rho_0, X_0 \rangle, \ldots, \langle \rho_n, X_n \rangle$ such that:

1. for every $0 \leq i \leq n$, $X_i \subseteq X$ and $\rho_i$ is a map from $C_i$ to $\mathcal{P}(X_i)$;
2. for every $0 \leq i < n$,
   a. if $\ell(t_{i+1}) \in \Sigma$, and if $\rho_0, p_1$ are respectively the input and output places of $t_{i+1}$, there is an element $x \in \rho_i(p_0)$ such that $\lambda(x) = \ell(t_{i+1})$ and:

$$X_{i+1} = X_i \setminus \{x\}; \quad \rho_{i+1}(p) = \begin{cases} \{y \in X_{i+1} \mid x \leq y\} & \text{if } p = p_1 \\ \rho_i(p) \setminus \{x\} & \text{otherwise.} \end{cases}$$
   
   b. if $\ell(t_{i+1}) = \tau$, then

$$X_{i+1} = X_i; \quad \rho_{i+1}(p) = \begin{cases} \bigcup_{q \in t_{i+1}} \rho_i(q) & \text{if } p \in t_{i+1}^* \\ \rho_i(p) & \text{otherwise.} \end{cases}$$

The reading is initial if $C_0 = \{p_{in}\}$ and $\rho_0(p_{in}) = X_0 = X$. The reading is final if $C_n = \{p_{fin}\}$ and $X_n = \emptyset$. The reading is accepting if it is both initial and final. $P$ is accepted by $\mathcal{N}$ if there is an accepting reading of $P$ in $\mathcal{N}$. The language recognised by $\mathcal{N}$ is the set of pomsets accepted by $\mathcal{N}$. It should not be confused with the pomset language of $\mathcal{N}$, as defined above.

Remark. Notice that, if $R$ is accepting, the existence of an accepting reading of $P$ along $R$ can be tested by a simple history-independent non-deterministic algorithm. We start with $X_0 = X$ and $\rho_0 = [p_{in} \mapsto X]$. At step $i+1$ we use condition 2b to compute $\rho_{i+1}$ and $X_{i+1}$ if $t_{i+1}$ is silent. If $t_{i+1}$ is visible and there is no $x \in \rho_i(t_{i+1})$ such that $\lambda(x) = \ell(t_{i+1})$, then we conclude that there are no readings of $P$ along $R$. Otherwise, we non-deterministically choose an appropriate $x$ and use condition 2a to compute $\rho_{i+1}$ and $X_{i+1}$. If this yields $X_n = \emptyset$ we have an accepting reading, otherwise we can conclude that there are no such readings.
Lemma 3. If $R$ is accepting, there is an accepting reading of $P$ along $R$ if and only if $P \sqsubseteq P(R)$.

Proof. See [3].

Corollary 4. The language recognised by $\mathcal{N}$ is $\subseteq [\mathcal{N}]$.

4 Rational Petri nets

This section shows that every rational pomset language is the pomset language of a (safe labelled) Petri net. To this end, we recursively associate with every expression $e$ a Petri net $\mathcal{N}(e)$ such that $[\mathcal{N}(e)] = [e]$. Moreover, all accepting runs of this Petri net turn out to be series-parallel. The construction poses no difficulty; it is a simple adaptation of Thompson’s construction for rational word languages [18], and an extension of a previous construction by Grabowski [6] for safe Petri nets and pomset languages. We only present this construction graphically here.

\[
\begin{align*}
\mathcal{N}(0) &= \quad [\quad] \\
\mathcal{N}(1) &= \quad [\quad] \\
\mathcal{N}(a) &= \quad [\quad] \\
\mathcal{N}(e_1 + e_2) &= \quad [\quad] \\
\mathcal{N}(e_1 \parallel e_2) &= \quad [\quad] \\
\mathcal{N}(e_1 \cdot e_2) &= \quad [\quad] \\
\mathcal{N}(e^*) &= \quad [\quad]
\end{align*}
\]

This construction yields decidability of biKA in exponential space. Indeed we may build the Petri nets $\mathcal{N}(e)$ and $\mathcal{N}(f)$ from the expressions $e$ and $f$ (these are linear in the size of $e$ and $f$) and use Jategaonkar and Meyer’s result [10] that testing containment of pomset-trace languages of two Petri nets is an ExpSpace-complete problem.

Theorem 5. The problem biKA lies in the class ExpSpace.

Proposition 6. The language recognised by $\mathcal{N}(e)$ is $\subseteq [e]$.

Proof. By construction we have $[\mathcal{N}(e)] = [e]$. We conclude using Corollary 4.

5 Comparing Petri nets modulo refinement

Next we show how to compare Petri nets modulo refinement. Thanks to the previous construction, this leads to decidability of the problem CKA. We fix two Petri nets $\mathcal{N}_1$ and $\mathcal{N}_2$ for this section and the following one. Our goal is to check whether $[\mathcal{N}_1] \subseteq [\mathcal{N}_2]$, i.e., whether for each run $R_1 \in \mathcal{L}(\mathcal{N}_1)$, there exists a corresponding run $R_2 \in \mathcal{L}(\mathcal{N}_2)$ such that $P(R_1) \sqsubseteq P(R_2)$.

The first difficulty is that we may have to reorder runs in $\mathcal{N}_2$: due to concurrency, transitions might be triggered in different orders and still yield the same pomset.
5.1 Reordering runs

Let $R = t_1; \ldots; t_n$ be a run from $C_0$ to $C_n$. Let $\pi$ be a permutation of $[1..n]$. The action of $\pi$ on $R$ is defined as $\pi R \triangleq t_{\pi(1)}; \ldots; t_{\pi(n)}$. The permutation $\pi$ is compatible with $R$ if it is order-preserving:

$$\forall i, j, i \leq_R j \Rightarrow \pi(i) \leq \pi(j).$$

Lemma 7. If $\pi$ is compatible with $R$, then $C_0 \xrightarrow{\pi R} C_n$, $G(R) = G(\pi R)$, and $i \leq_R j$ if $\pi(i) \leq_R \pi(j)$.

Proof. We can exchange two successive transitions that are not causally linked without changing the graph (up-to isomorphism). We repeat this process until we obtain $\pi R$.

Accordingly, we say that a run $R'$ is equivalent to a run $R$ if $R' = \pi R$ for some compatible permutation $\pi$.

Another important notion for the completeness of the method we propose is that of an economical run: a run that fires its silent transitions as late as possible.

Definition 8. A run $t_1; \ldots; t_n$ is economical if for all $i < j$, if $t_i, \ldots, t_{j-1}$ are silent transitions and $t_j$ is a visible transition, then $i \leq_R j$.

The run in Figure 2 is not economical: the fifth transition is a silent one, but it is not causally related to the next transition, which is visible. We will see that it can be reordered into an equivalent economical run (Proposition 10 and Example 11 below.)

Even more importantly when comparing two runs, we need to ensure that the visible transitions are fired in the same order.

Definition 9. Given a run $R_1$ in $N_1$ and a run $R_2$ in $N_2$ such that $\mathbb{P}(R_1) \subseteq \mathbb{P}(R_2)$, we say that $R_2$ follows $R_1$ if the subsumption is witnessed by a bijection $\varphi$ such that for every two visible indices $i, j$ in $R_2$ we have $i < j \Leftrightarrow \varphi(i) < \varphi(j)$.

Proposition 10. Let $R_1$ and $R_2$ be series-parallel runs in $N_1$ and $N_2$, respectively. If $\mathbb{P}(R_1) \subseteq \mathbb{P}(R_2)$ then there exists an economical and series-parallel run $R'_2$ in $N_2$ that follows $R_1$ and is equivalent to $R_2$.

Example 11. Consider the run $R_2$ from Figure 2 and the following run $R_1$:

The pomset of $R_1$ is $\mathbb{P}(R_1) = P_c \parallel P_d$, and we may check that $\mathbb{P}(R_1) \subseteq \mathbb{P}(R_2)$. To transform $R_2$ into a run that is economical and follows $R_1$, we must (1) exchange the transitions labelled with $c$ and $d$; and (2) delay the silent transition in the middle of $R_2$ until all visible transitions have been fired. Doing so, we get the following run $R'_2$, which follows $R_1$ and is equivalent to $R_2$. 

$$R'_2 : \hspace{1cm} 1 \rightarrow 2 \rightarrow 4 \rightarrow 6 \rightarrow 8 \rightarrow 9 \rightarrow 10 \rightarrow 11$$

$$R_1 : \hspace{1cm} 1 \rightarrow 2 \rightarrow 4 \rightarrow 6 \rightarrow 8 \rightarrow 9 \rightarrow 10 \rightarrow 11$$
5.2 Loci

In this more technical section, we define loci, as a way to lift the causality relation between transitions to places in the successive configurations of the runs. Let $R$ be a run, with $R = C_0 \xrightarrow{i_0} C_1 \cdots \xrightarrow{i_{n-1}} C_n$, $T(R) = ([1..n], \preceq_R, \lambda)$. A locus denotes a pair $\langle p, i \rangle$ where $0 \leq i \leq n$ and $p \in C_i$. In some sense, loci are places with a time index. In the previous pictures those are the numbered circles: $p$ is the number in the circle (the name of the place), and $i$ is the index of its column. Formally, the set of loci of the run $R$ is $\bigcup_{0 \leq i \leq n} C_i \times \{i\}$. We generate an equivalence relation $\approx_R$ on loci using the following rule:

$$p \notin t_i \Rightarrow \langle p, i \rangle \approx_R \langle p, i - 1 \rangle.$$  

Graphically, equivalent loci are linked with dotted lines. Equivalences classes with respect to $\approx_R$ are thus places with a time interval, rather than a single index. The source of a locus $\langle p, i \rangle$ is the smallest index of its equivalence class:

$$\text{src} \langle p, i \rangle \triangleq \min \{ j \leq i \mid \langle p, i \rangle \approx_R \langle p, j \rangle \}.$$  

Now we define a preorder $\preceq_R$, generated by the following rules:

$$p, q \in t_i \times t_i^* \Rightarrow \langle p, i - 1 \rangle \preceq_R \langle q, i \rangle, \quad \langle p, i \rangle \approx_R \langle q, i \rangle \Rightarrow \langle p, i \rangle \preceq_R \langle q, j \rangle.$$  

Note that two loci in the same configuration are always incomparable. Finally, we inductively define the set of indices of predecessors of a locus:

- $\text{pred} (p, 0) \triangleq \emptyset$.
- if $p \in t_i^*$, then $\text{pred} (p, i) \triangleq \{ i \} \cup \bigcup_{q \in t_i^*} \text{pred} (q, i - 1)$.
- if $p \notin t_i^*$, then $\text{pred} (p, i) \triangleq \text{pred} (p, i - 1)$.

The set of visible predecessors of a locus, written $\text{vpred} (p, i)$, is the subset of indices of visible transitions in $\text{pred} (p, i)$.

Lemma 12. The following properties hold:

\begin{align*}
\forall i, \forall p \in t_i^*, \quad & \text{pred} (p, i) = \{ j \mid j \leq_R i \} \quad (1) \\
\forall i, \forall p \in C_i, \quad & \text{pred} (p, i) = \{ j \mid j \leq_R \text{src} (p, i) \} \quad (2) \\
\forall i, j, p, q, \quad & \langle p, i \rangle \approx_R \langle q, j \rangle \Rightarrow \text{pred} (p, i) = \text{pred} (q, j) \quad (3) \\
\forall i, j, p, q, \quad & \langle p, i \rangle \preceq_R \langle q, j \rangle \Rightarrow \text{pred} (p, i) \subseteq \text{pred} (q, j) \quad (4) \\
\forall i, j, p, \forall q \in t_i^*, \quad & j \in \text{vpred} (p, i) \Rightarrow \langle q, j \rangle \preceq_R \langle p, i \rangle \quad (5) \\
\forall i \leq j, \quad & \exists p \in C_j : i \in \text{pred} (p, j) \quad (6)
\end{align*}

Notice that we managed to lift the causality relation $\leq_R$ to the level of loci: this lemma implies that if $i \neq j$, $p \in t_i^*$ and $q \in t_j^*$, then $i \leq_R j$ if and only if $\langle p, i \rangle \preceq_R \langle q, j \rangle$.

5.3 Schedules

We compare runs using the following notion of schedule, where we interleave two runs in such a way that they synchronise on visible transitions.

Definition 13. Let $R_1$ and $R_2$ be two runs with $R_i = C_0^i \xrightarrow{i_0^i} C_1^i \cdots \xrightarrow{i_{n_{i-1}}^i} C_{n_i}$ for $i \in \{1, 2\}$. An $N$-schedule from $R_1$ to $R_2$ is a function $\eta : [0..N] \to [0..n_1] \times [0..n_2]$ such that

- $\eta(0) = (0, 0)$ and $\eta(N) = (n_1, n_2)$;
- if $\eta(k) = (i, j)$, then either
1. $t^1_{i+1}$ is a silent transition and $\eta(k+1) = \langle i+1, j \rangle$, or
2. $t^2_{j+1}$ is a silent transition and $\eta(k+1) = \langle i, j+1 \rangle$, or
3. $t^1_{i+1}$ and $t^2_{j+1}$ are visible, $\ell(t^1_{i+1}) = \ell(t^2_{j+1})$ and $\eta(k+1) = \langle i+1, j+1 \rangle$.

Note that a schedule constructs a bijection between the visible transitions of $R_1$ and those of $R_2$. Indeed, each of these transitions must be fired synchronously and agree on labels (case 3).

Furthermore, since a schedule starts with $\eta(0) = \langle 0, 0 \rangle$ and ends with $\eta(N) = \langle n_1, n_2 \rangle$, every transition in both runs must be fired. This means there is a label-preserving bijection $\varphi$, called the bijection induced by $\eta$, from the visible transitions of $R_2$ to those of $R_1$ that satisfies $i < j$ if and only if $\varphi(i) < \varphi(j)$.

The notion of schedule is still very weak: there are schedules from one run to another whenever they have the same visible transitions, in the same order. Causality between those transitions is not taken into account. We fix this with the following technical definition.

Intuitively, we keep track of the history and relationships between the loci of the configurations, in order to ensure that the causality relation in the presumably smaller run $R_1$ refines that of $R_2$.

**Definition 14.** For each $N$-schedule we define the following sequence of binary relations $(\prec_k)_{k \in \{0 \ldots N\}}$, where $\prec_k \subseteq C_i^1 \times C_j^2$ when $\eta(k) = \langle i, j \rangle$, by induction:

1. if $\eta(k+1) = \langle i+1, j \rangle$, we set $p \prec_{k+1} q \triangleq \begin{cases} p \prec_k q & \text{if } p \notin (t^1_{i+1})^* \\ \exists q' \in (t^1_{i+1})^*: p \prec_k q' & \text{otherwise.} \end{cases}$
2. if $\eta(k+1) = \langle i, j+1 \rangle$, we set $p \prec_{k+1} q \triangleq \begin{cases} p \prec_k q & \text{if } q \notin (t^2_{j+1})^* \\ \forall q' \in (t^2_{j+1})^*: p \prec_k q' & \text{otherwise.} \end{cases}$
3. otherwise, let $t^1_{i+1} = \langle \{p_0\}, a, \{p_1\} \rangle$ and $t^2_{j+1} = \langle \{q_0\}, a, \{q_1\} \rangle$; we set $p \prec_{k+1} q \triangleq \begin{cases} p_0 \prec_k q \text{ or } (q = q_1 \text{ and } p_0 \prec_k q_0) & \text{if } p = p_1 \\ p \prec_k q \text{ and } q \neq q_1 & \text{otherwise.} \end{cases}$

The schedule $\eta$ is valid if for every visible index $i$ in $R_2$ we have $p \prec_k q$ for the unique $k, p, q$ such that $\eta(k) = \langle \varphi(i), i \rangle$, $(t^1_{\varphi(i)})^* = \{p\}$ and $(t^2_j)^* = \{q\}$.

**Example 15.** Recall the runs $R_2^1$ and $R_1$ from Example 11. The following sequence is a schedule from $R_1$ to $R_2^1$.

$$\eta = (0, 0); (1, 0); (1, 1); (2, 2); (3, 3); (4, 4); (5, 5); (6, 5); (6, 6); (6, 7); (6, 8)$$

We may then draw the two runs side by side according to this schedule:

![Diagram](image-url)
Remark. If $\eta(k) = \langle i, j \rangle$, $\eta(k') = \langle i', j' \rangle$, $\langle p, i \rangle \approx_{R_1} \langle p', i' \rangle$ and $\langle q, j \rangle \approx_{R_2} \langle q', j' \rangle$, then $p \prec_k q$ if and only if $p' \prec_{k'} q'$.

Lemma 16. If $\eta(k) = \langle i, j \rangle$, then $p \prec_k q$ entails $\varphi(\text{vpred}(q, j)) \subseteq \text{vpred}(p, i)$.

The algorithm we define in the next section looks for valid schedules. The following proposition establishes soundness of this strategy.

Proposition 17. If there exists a valid schedule from $R_1$ to $R_2$, then $\mathbb{P}(R_1) \sqsubseteq \mathbb{P}(R_2)$.

Proof. The bijection $\varphi$ induced by $\eta$ works. Let $i, j$ be visible indices in $R_2$ such that $i \preceq_R j$. We need to show that $\varphi(i) \preceq_{R_1} \varphi(j)$. Take the unique $k, p, q$ such that $\eta(k) = \langle \varphi(j), j \rangle$, $(t_{\varphi(j)}^k)^* = \{p\}$ and $(t_j^q)^* = \{q\}$. By Lemma 12.1, we have $i \in \text{vpred}(q, j)$. Since $\eta$ is valid, we have $p \prec_k q$, and thus $\varphi(\text{vpred}(q, j)) \subseteq \text{vpred}(p, \varphi(j))$ by Lemma 16. This means that $\varphi(i) \in \text{vpred}(p, \varphi(j))$, and using Lemma 12.1 again we obtain $\varphi(i) \preceq_{R_1} \varphi(j)$.

For completeness of the algorithm we need to exhibit valid schedules. Under appropriate assumptions—see Proposition 19 below—the following canonical schedule $\eta$ from $R_1$ to $R_2$ will work. We define it recursively. Intuitively, we schedule the silent transitions of $R_1$ as early as possible and those of $R_2$ as late as possible:

- $\eta(0) = (0, 0)$;
- if $\eta(k) = (i, j)$ then
  1. if $t_i^{i+1}$ is silent, then $\eta(k + 1) = (i + 1, j)$;
  2. if $t_i^{i+1}$ is visible and $t_j^{j+1}$ is silent then $\eta(k + 1) = (i, j + 1)$;
  3. if $t_i^{i+1}$ and $t_j^{j+1}$ are visible, then $\eta(k + 1) = (i + 1, j + 1)$.

We write $\varphi$ for the bijection induced by $\eta$. The schedule from Example 15 is actually the canonical schedule. The converse of Lemma 16 holds for the canonical schedule:

Lemma 18. If $R_2$ is series-parallel and economical, then, for every $k$ with $\eta(k) = \langle i, j \rangle$, if $\varphi(\text{vpred}(q, j)) \subseteq \text{vpred}(p, i)$, then $p \prec_k q$.

Proposition 19. If $R_1$ and $R_2$ are series-parallel, if $\mathbb{P}(R_1) \sqsubseteq \mathbb{P}(R_2)$, and if $R_2$ is economical and follows $R_1$, then the canonical schedule $\eta$ from $R_1$ to $R_2$ is valid.

Proof. First note that since $R_2$ follows $R_1$, $\varphi$ and the bijection witnessing $\mathbb{P}(R_1) \sqsubseteq \mathbb{P}(R_2)$ must coincide. Let $i$ be a visible index in $R_2$, and let $k, p, q$ such that $\eta(k) = \langle \varphi(i), i \rangle$, $(t_{\varphi(i)}^k)^* = \{p\}$ and $(t_i^q)^* = \{q\}$. We have to prove $p \prec_k q$. By Lemma 18, it suffices to prove the inclusion $\varphi(\text{vpred}(q, i)) \subseteq \text{vpred}(p, \varphi(i))$, i.e., that for every $j \in \text{vpred}(q, i)$, we have $\varphi(j) \in \text{vpred}(p, \varphi(i))$. This is equivalent to checking that for every visible $j$, $j \preceq_{R_2} i$ implies $\varphi(j) \preceq_{R_1} \varphi(i)$, which is true because $\varphi$ is an order preserving bijection from the pomsets of $R_2$ to that of $R_1$.

Note that this lemma relies on the fact that $R_2$ is series-parallel. Indeed, there can be pairs of runs $R_1$ and $R_2$ satisfying $\mathbb{P}(R_1) \sqsubseteq \mathbb{P}(R_2)$, and $R_2$ being economical and following $R_1$, but such that the canonical schedule is not valid.

5.4 Reduction to finite automata

Now that we have the notion of valid schedule, we use a technique similar to [10] to reduce the problem of comparing Petri nets modulo subsumption to the comparison of plain automata. For this end, we define the following automaton that aims at recognising those runs of $\mathcal{N}_1$ for which there exists a valid schedule to some run in $\mathcal{N}_2$. CONCUR 2017
Lemma 21. The language of the automaton $\mathcal{N}_1 \prec \mathcal{N}_2$ is the set of accepting runs $R_1$ in $\mathcal{N}_1$ such that there is an accepting run $R_2$ in $\mathcal{N}_2$ and a valid schedule from $R_1$ to $R_2$.

Finally, we can reduce the comparison of Petri nets modulo subsumption to that of (word) automata.

Proposition 22. If the runs in $\mathcal{N}_1$ and those in $\mathcal{N}_2$ are all series-parallel, then $\mathcal{L}(\mathcal{A}(\mathcal{N}_1)) \subseteq \mathcal{L}(\mathcal{N}_1 \prec \mathcal{N}_2)$ if and only if $[\mathcal{N}_1] \subseteq [\mathcal{N}_2]$.

Proof. Suppose $\mathcal{L}(\mathcal{A}(\mathcal{N}_1)) \subseteq \mathcal{L}(\mathcal{N}_1 \prec \mathcal{N}_2)$ and let $P \in [\mathcal{N}_1]$. There exists $R_1 \in \mathcal{L}(\mathcal{A}(\mathcal{N}_1))$ such that $P = \mathbb{P}(R_1)$. By assumption we also have $R_1 \in \mathcal{L}(\mathcal{N}_1 \prec \mathcal{N}_2)$, which means, by Lemma 21, that there is an accepting run $R_2$ in $\mathcal{N}_2$ and a valid schedule $\eta$ from $R_1$ to $R_2$. Proposition 17 then tells us that $\mathbb{P}(R_1) \subseteq \mathbb{P}(R_2)$, thus proving $P \in [\mathcal{N}_2]$.

Conversely, assume that $[\mathcal{N}_1] \subseteq [\mathcal{N}_2]$. Let $R_1 \in \mathcal{L}(\mathcal{A}(\mathcal{N}_1))$ be an accepting run in $\mathcal{N}_1$. By assumption, there is an accepting run $R_2$ in $\mathcal{N}_2$ such that $\mathbb{P}(R_1) \subseteq \mathbb{P}(R_2)$. By hypothesis, both $R_1$ and $R_2$ are series-parallel. By Proposition 10, there exists an economical series-parallel run $R'_2$ that follows $R_1$ and is equivalent to $R_2$. Hence using Proposition 19, there is a valid schedule from $R_1$ to $R'_2$. With Lemma 21 we conclude that $R_1 \in \mathcal{L}(\mathcal{N}_1 \prec \mathcal{N}_2)$. ▶
6 Decidability & Complexity of CKA

Putting together the results from Sections 4 and 5 yields the announced algorithm.

Proposition 23. CKA lies in the class ExpSpace.

Proof. We build the Petri nets $N(e)$ and $N(f)$, and then the finite automata $A(N(e))$ and $N(e) \prec N(f)$. By Proposition 22, to answer the original question, we simply need to test these automata for language inclusion. It is well known that this requires polynomial space with respect to the size of the automata.

Let $n$ be the size of $e$ and $m$ the size of $f$ (their number of symbols). The Petri nets $N(e)$ and $N(f)$ are linear in the size of $e$ and $f$, with at most $2n$ and $2m$ places. The automaton $A(N(e))$ uses at most $2^{2n}$ states (recall these are sets of places). The automaton $N(e) \prec N(f)$ uses at most $2^{2n} \times 2^{2m} \times 2^{2n \times 2m}$ states. Hence testing language equivalence of these two automata will use an amount of space polynomial in $2^{2n}$ and $2^{2n+2m+4mn}$, whence the announced result.

The problem of testing whether the language of a regular expression with shuffle is equal to $\Sigma^*$ is ExpSpace-complete.

The key observations for our reduction are due to Grabowski [6]: words are isomorphic to totally ordered pomsets, and given two words $u$ and $v$, the set of totally ordered pomsets in $\{u \parallel v\}$ is isomorphic to the shuffle product of $u$ and $v$.

Concretely, we associate a series-parallel expression $[e]$ to any regular expression with shuffle $e$ by replacing every occurrence of $\parallel$ with $\parallel$. This encoding has the following property.

Lemma 25. For every word $w \in \Sigma^*$, we have $w \in [e]$ if and only if $w$ seen as a totally ordered pomset is in $\{[e]\}$.

Proof. By a simple induction on $e$, using the above observation for the shuffle case. (Each subcase can be found in [6].)

As a consequence, the language of $e$ is $\Sigma^*$ if and only if $\{\Sigma^*\} \subseteq \{[e]\}$. We thus have a linear encoding of the universality of regular expressions with shuffle into containment of downward-closed rational languages, hence our final theorem.


An implementation of the algorithm is available at http://paul.brunet-zamansky.fr/cka.html.
7 Related work

Several constructions in the literature are similar to those presented in Section 4. Here we list some of them, highlighting the differences between these developments and our own.

Lodaya and Weil introduced branching automata that recognise series-parallel rational pomset languages [15], which include the series-rational languages we use here. These automata impose a strong notion of bracketing (opening and closing $\tau$-transitions must match exactly), which we do not know how to handle when it comes to comparing automata. This is why we used plain Petri nets instead.

Jategaonkar and Meyer presented a construction almost equivalent to ours [10], albeit for different purposes: their goal was to obtain a lower complexity bound by a reduction from the universality problem of regular languages with shuffle. The main differences in the constructions are that we use an initial place instead of an initial marking, and that we consider a unique final place while they have a distinguished final transition. These differences mainly impact the star and parallel product constructs. Jategaonkar and Meyer’s construction could in fact be adapted to obtain an alternative proof of Theorem 5. However, their construction does not satisfy the structural constraints needed for the completeness of the algorithm we develop in Section 5: the runs of the automata produced by their construction are not always series-parallel.

Finally, a third construction that produces safe Petri nets from expressions was developed by Lodaya [14]. It is, however, quite different from the present approach. In particular, it requires initial and final markings, and it is not appropriate for a precise complexity analysis, as it produces nets that are exponentially large with respect to input expressions.

References


3 Paul Brunet, Damien Pous, and Georg Struth. On decidability of concurrent Kleene algebra, 2017. Full version of this extended abstract, with proofs. URL: https://hal.archives-ouvertes.fr/hal-01558108/.


Model-Checking Counting Temporal Logics on Flat Structures*†

Normann Decker¹, Peter Habermehl², Martin Leucker³, Arnaud Sangnier⁴, and Daniel Thoma⁵

1 ISP, University of Lübeck, Lübeck, Germany
decker@isp.uni-luebeck.de
2 IRIF, Univ. Paris Diderot, Paris, France
haberm@irif.fr
3 ISP, University of Lübeck, Lübeck, Germany
leucker@isp.uni-luebeck.de
4 IRIF, Univ. Paris Diderot, Paris, France
sangnier@irif.fr
5 ISP, University of Lübeck, Lübeck, Germany
thoma@isp.uni-luebeck.de

Abstract
We study several extensions of linear-time and computation-tree temporal logics with quantifiers that allow for counting how often certain properties hold. For most of these extensions, the model-checking problem is undecidable, but we show that decidability can be recovered by considering flat Kripke structures where each state belongs to at most one simple loop. Most decision procedures are based on results on (flat) counter systems where counters are used to implement the evaluation of counting operators.

1998 ACM Subject Classification D.2.4 Software/Program Verification

Keywords and phrases Counting Temporal Logic, Model checking, Flat Kripke Structure

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2017.29

1 Introduction

Model checking [8] is a method to verify automatically the correct behaviour of systems. It takes as input a model of the system to be verified and a logical formula encoding the specification and checks whether the behaviour of the model satisfies the formula. One key aspect of this method is to find the appropriate balance between expressiveness of models and logical formalisms and efficiency of the model-checking algorithms. If the model is too expressive, e.g., Turing machines, then the model-checking problem, even with very simple logical formalisms, becomes undecidable. On the other hand, some expressive logics have been proposed in order to reason on the temporal executions of simple models such as Kripke structures. This is the case for the linear temporal logic LTL [22] and the branching-time temporal logics CTL [7] and CTL* [14], for which the model-checking problem has been shown to be PSPACE-complete, contained in P and PSPACE-complete, respectively (see, e.g., [3]).

* Partially supported by EGIDE/DAAD-Procope (FREQS) and European Commission H2020 Project COEMS.
† An extended version is available [9] providing additional proof details.

© Normann Decker, Peter Habermehl, Martin Leucker, Arnaud Sangnier, and Daniel Thoma; licensed under Creative Commons License CC-BY
Even though these logical formalisms allow for stating classical properties like safety or liveness over executions of Kripke structures, their expressiveness is limited. In particular they cannot describe quantitative aspects, as for instance the fact that a property has been true twice as often as another along an execution. One approach to solve this issue is to extend the logic with some ability to count positions of an execution satisfying some property and to check constraints over such numbers at some positions. Such a counting extension is proposed in [19] for CTL leading to a logic denoted here as $\text{cCTL}$. This formalism can state properties such as an event $p$ will eventually occur and before that, the number of events $q$ is larger than two. The authors propose further an extension called (here) $\text{cCTL}_\pm$ that admits diagonal comparisons (i.e., negative and positive coefficients) to state, for instance that the number of events $b$ is greater than the number of events $c$. It is shown that the model-checking problem for $\text{cCTL}$ is decidable in polynomial time and that the satisfiability problem for $\text{cCTL}_\pm$ is undecidable. A similar extension for LTL is considered in [18] where it is proven that model checking of $\text{cLTL}$ is ExpSpace-complete while that of $\text{cLTL}_\pm$ is undecidable.

Following the same motivation, regular availability expressions (RAE) were introduced in [16] extending regular expressions by a mechanism to express that on a (sub-)word matching an expression specific letters occur with a given relative frequency. Unfortunately, emptiness of the intersection of two such expressions was shown undecidable. Even for single expressions only a non-elementary procedure is known for verification (inclusion in regular languages) and deciding emptiness [1]. The case is similar for the logic $\text{fLTL}$ [5], a variant of LTL that features an until operator extended by a frequency constraint. The operator is intended to relax the classical semantics where $\varphi \mathbf{U} \psi$ requires $\varphi$ to hold at all positions before $\psi$. For example, the $\text{fLTL}$ formula $p \mathbf{U} q$ states that $q$ holds eventually and before that the proportion of positions satisfying $p$ should be at least one third. The concept of relative frequencies embeds naturally into the context of counting logics as it can be understood as a restricted form of counting. In fact, $\text{fLTL}$ can be considered as a fragment of $\text{cLTL}_\pm$ and still has an undecidable satisfiability problem [5] implying the same for model-checking Kripke structures. Moreover, most techniques employed for obtaining results on RAE as well as $\text{fLTL}$ involve variants of counter systems.

Looking at the model-checking problem from the model point of view, recent work has shown that restrictions can be imposed on Kripke structures to obtain better complexity bounds. As a matter of fact if the structure is flat (or weak), which means every state belongs to at most one simple cycle in the graph underlying the structure, then the model-checking problem for LTL becomes NP-complete [17]. Such a restriction has as well been successfully applied to more complex classes of models. It is well known that the reachability problem for two-counter systems is undecidable [21] whereas for flat systems the problem is decidable for any number of counters [15], even more, model checking of LTL is NP-complete [11]. Flat structures are not only interesting because of their algorithmic properties, but also because they can be used as a way to under-approximate the behaviour of non-flat systems. For instance for counter systems one gets a semi-decision procedure for the reachability problem which consists in enumerating flat sub-systems and testing for reachability. In simple words, flat structures can be understood as an extension of paths typically used in bounded model checking and we expect that bounded model checking using flat structures rather than paths improves practical model checking approaches.

**Contributions.** We consider the model-checking problem for a counting logic that we call $\text{CCTL}^*$ where we use variables to mark positions on a run from where we begin to count the
Table 1 Complexity characterisation of the model-checking problems of fragments of CCTL*. PH indicates polynomial reducibility to the (decidable) satisfiability problem of PH.

<table>
<thead>
<tr>
<th></th>
<th>CTL</th>
<th>LTL</th>
<th>CTL*</th>
<th>fLTL</th>
<th>fCTL</th>
<th>fCTL*</th>
<th>CLTL</th>
<th>CCTL</th>
<th>CCTL*</th>
</tr>
</thead>
<tbody>
<tr>
<td>FKS</td>
<td>P</td>
<td>NP-c. [17]</td>
<td>PSPACE</td>
<td>NExp</td>
<td>Exp</td>
<td>ExpSpace</td>
<td>PH</td>
<td>PH</td>
<td>PH</td>
</tr>
</tbody>
</table>

number of times a subformula is satisfied. Such a way of counting was also introduced in [19], see Section 2.2 for a comparison. We study as well its fragments fCTL, fLTL and fCTL* where the explicit counting mechanism is replaced by a generalized version of the until operator capable of expressing frequency constraints.

First we prove that fCTL model checking is at most exponential in the formula size and polynomial in the structure size by using an algorithm similar to the one for CTL model checking. To deal with frequency constraints a counter is employed for tracking the number of times a subformula is satisfied in a run of a Kripke structure. We then show that for flat Kripke structures the model-checking problems of fLTL and CCTL* are decidable. For the former, our method is a guess and check procedure based on the existence of a flat counter system as witness of a run of the Kripke structure satisfying the fLTL formula. For the latter, we use a technique which consists in encoding the run of a flat Kripke structure into a Presburger arithmetic formula and then we show that model checking of CCTL* can be translated into the satisfiability problem of a decidable extension of Presburger arithmetic, called PH, featuring a counting quantifier known as Härtig quantifier. We hence provide new decidability results for CCTL* which in practice could be used as an under-approximation approach to the general model-checking problem. We furthermore relate an extension of Presburger arithmetic, for which the complexity of the satisfiability problem is open, to a concrete model-checking problem. In summary, for model checking different fragments of CCTL* on Kripke structures (KS) or flat Kripke structures (FKS) we obtain the picture shown in Table 1 where bold entries are our novel results.

2 Definitions

2.1 Preliminaries

We write \( \mathbb{N} \) and \( \mathbb{Z} \) to denote the sets of natural numbers (including zero) and integers, respectively, and \([i, j]\) for \( \{ k \in \mathbb{Z} \mid i \leq k \leq j \} \). We consider integers encoded with a binary representation. For a finite alphabet \( \Sigma \), \( \Sigma^* \) represents the set of finite words over \( \Sigma \), \( \Sigma^+ \) the set of finite non-empty words over \( \Sigma \) and \( \Sigma^\omega \) the set of infinite words over \( \Sigma \). For a finite set \( E \) of elements, \( |E| \) represents its cardinality. For (finite or infinite) words and general sequences \( u = a_0 a_1 \ldots a_k \ldots \) of length at least \( k + 1 > 0 \) we denote by \( u(k) = a_k \) the \((k+1)\)-th element and refer to its indices \( 0, 1, \ldots \) as positions on \( u \). If \( u \) is finite then \( |u| \) denotes its length. For arbitrary functions \( f : A \to B \) and elements \( a \in A, b \in B \) we denote by \( f[a \mapsto b] \) the function \( f' \) that is equal to \( f \) except that \( f'(a) = b \). We write 0 and 1 for the functions \( f_0 : A \to \{0\} \) and \( f_1 : A \to \{1\} \), respectively, if the domain \( A \) is understood. By \( B^A \) for sets \( A \) and \( B \) we denote the set of all functions from \( A \) to \( B \).
Kripke structures. Let \( AP \) be a finite set of atomic propositions. A Kripke structure is a tuple \( K = (S, s_I, E, \lambda) \) where \( S \) is a finite set of control states, \( s_I \in S \) the initial control state, \( E \subseteq S \times S \) the set of edges and \( \lambda : S \to 2^{AP} \) the labelling function. A finite path in \( K \) is a sequence \( u = s_0 s_1 \ldots s_k \in S^+ \) with \((s_i, s_{i+1}) \in E \) for all \( i \in [0, k - 1] \). Infinite paths are defined analogously. A run \( \rho \) of \( K \) is an infinite path with \( \rho(0) = s_I \). We denote by \( \text{Runs}(K) \) the set of runs of \( K \). Due to the single initial state, we assume without loss of generality that the graph of control states underlying \( K \) is connected, i.e. all states are reachable. A simple loop in \( K \) is a finite path \( u = s_0 s_1 \ldots s_k \) such that \( i \neq j \) implies \( s_i \neq s_j \) for all \( i, j \in [0, k] \) and \((s_k, s_0) \in E \). A Kripke structure \( K \) is called flat if for each state \( s \in S \) there is at most one simple loop \( u \) in \( K \) with \( u(0) = s \). See Figure 1 for an example. The classes of all Kripke structures and all flat Kripke structures are denoted \( KS \) and \( FKS \), respectively.

Counter systems. Our proofs use systems with integer counters and simple guards. A counter system is a tuple \( S = (S, s_I, C, \Delta) \) where \( S \) is a finite set of control states, \( s_I \in S \) is the initial state, \( C \) is a finite set of counter names and \( \Delta \subseteq S \times \mathbb{Z}^C \times 2^{\Theta(C)} \times S \) is the transition relation where \( \Theta(C) = \{(c < 0), (c \geq 0) \mid c \in C \} \). An infinite sequence \( s_0 s_1 \ldots \in S^\omega \) of states starting in \( s_0 = s_I \) is called a run of \( S \) if there is a sequence \( \theta_0 \theta_1 \ldots \in (\mathbb{Z}^C)^\omega \) of valuation functions \( \theta_i : C \to \mathbb{Z} \) with \( \theta_0 = 0 \) and a transition \((s_i, u_i, G_i, s_{i+1}) \in \Delta \) for every \( i \in \mathbb{N} \) such that \( \theta_{i+1} = \theta_i + u_i \) (defined point-wise as usual), \( \theta_{i+1}c < 0 \) if \( (c < 0) \in G_i \) and \( \theta_{i+1}c \geq 0 \) if \( (c \geq 0) \in G_i \) for all \( c \in C \). Again, we denote by \( \text{Runs}(S) \) the set of all such runs and assume the graph of control states underlying \( S \) is connected.

2.2 Temporal Logics with Counting

We now introduce the different formalisms we use in this work as specification language. The most general one is the branching-time logic \( \text{CCTL}^* \) which extends the branching-time logic \( \text{CTL}' \) (see e.g. [3]) with the following features: it has operators that allow for counting along a run the number of times a formula is satisfied and which stores the result into a variable. The counting starts when the associated variable is “placed” on the run. These variables may be shadowed by nested quantification, similar to the semantics of the freeze quantifier in linear temporal logic [13].

Let \( V \) be a set of variables and \( AP \) a set of atomic propositions. The syntax of \( \text{CCTL}^* \) formulae \( \varphi \) over \( V \) and \( AP \) is given by the grammar rules

\[
\varphi ::= p \mid \varphi \land \varphi \mid \neg \varphi \mid X \varphi \mid \varphi U \varphi \mid E \varphi \mid x.\varphi \mid \tau \leq \tau \mid \tau ::= a \mid a \cdot \#_x(\varphi) \mid \tau + \tau
\]

for \( p \in AP, x \in V \) and \( a \in \mathbb{Z} \). Common abbreviations such as \( \top \equiv p \lor \neg p, \bot \equiv \neg \top, F \varphi \equiv \top U \varphi, G \varphi \equiv \neg F \neg \varphi \) and \( A \varphi \equiv \neg E \neg \varphi \) may also be used. The set of all subformulae of a formula \( \varphi \) (including itself) is denoted \( \text{sub}(\varphi) \) and \( |\varphi| \) denotes the length of \( \varphi \), with binary encoding of numbers.

Semantics. Intuitively, a variable \( x \) is used to mark some position on the concerned run. Within the scope of \( x \) a term \( \#_x(\varphi) \) refers to the number of times the formula \( \varphi \) holds between the current position and that marked by \( x \). The semantics of \( \text{CCTL}^* \) is hence defined with respect to a Kripke structure \( K = (S, s_I, E, \lambda) \), a run \( \rho \in \text{Runs}(K) \), a position \( i \in \mathbb{N} \) on \( \rho \) and a valuation function \( \theta : V \to \mathbb{N} \) assigning a position (index) on \( \rho \) to each variable. The
satisfaction relation $|=\text{ is defined inductively for } p \in AP$, formulae $\varphi, \psi$ and terms $\tau_1, \tau_2$ by

$(\rho, i, \theta) \models p \quad \overset{\text{def}}{=} \quad p \in \lambda(\rho(i)),$

$(\rho, i, \theta) \models X \varphi \quad \overset{\text{def}}{=} \quad (\rho, i + 1, \theta) \models \varphi,$

$(\rho, i, \theta) \models \varphi U \psi \quad \overset{\text{def}}{=} \quad \exists k \geq i : (\rho, k, \theta) \models \psi$ and $\forall j \in [i, k - 1] : (\rho, j, \theta) \models \varphi,$

$(\rho, i, \theta) \models E \varphi \quad \overset{\text{def}}{=} \quad \exists \rho' \in \text{Runs}(K) : \forall j \in [0, i] : \rho'(j) = \rho(j)$ and $\rho', i, \theta) \models \varphi,$

$(\rho, i, \theta) \models x. \varphi \quad \overset{\text{def}}{=} \quad (\rho, i, [x \mapsto i]) \models \varphi,$

$(\rho, i, \theta) \models \tau_1 \leq \tau_2 \quad \overset{\text{def}}{=} \quad [\tau_1](\rho, i, \theta) \leq [\tau_2](\rho, i, \theta),$

where the Boolean cases are omitted and the semantics of terms is given, for $a \in \mathbb{Z}$, by

\[
[a](\rho, i, \theta) \overset{\text{def}}{=} a,
\]

\[
[\tau_1 + \tau_2](\rho, i, \theta) \overset{\text{def}}{=} [\tau_1](\rho, i, \theta) + [\tau_2](\rho, i, \theta),
\]

\[
[a \cdot \#_x(\varphi)](\rho, i, \theta) \overset{\text{def}}{=} a \cdot [\{j \in N \mid \theta(x) \leq i, (\rho, j, \theta) \models \varphi\}].
\]

We abbreviate $(\rho, i, 0) \models \varphi$ by $(\rho, i) \models \varphi$ and $(\rho, 0) \models \varphi$ by $\rho \models \varphi$ and say that $\rho$ satisfies $\varphi$ (at position $i$) in these cases. Moreover, we say a state $s \in S$ satisfies $\varphi$, denoted $s \models \varphi$ if there are $\rho_s \in \text{Runs}(K)$ and $i \in N$ such that $\rho_s(i) = s$ and $(\rho_s, i) \models \varphi$. The Kripke structure $K$ satisfies $\varphi$, denoted by $K \models \varphi$. Note that we choose to define the model-checking relation existentially but since the formalism is closed under negation, this does not have major consequences on our results.

**Fragments.** We define the following fragments of $\text{CCTL}^*$ in analogy to the classical logics LTL and CTL. The linear time fragment $\text{CTL}^*$ consists of those $\text{CCTL}^*$ formulae that do not use the path quantifiers $E$ and $A$. The branching time logic $\text{CTT}$ restricts the use of temporal operators $X$ and $U$ such that each occurrence must be preceded immediately by either $E$ or $A$. Similar branching-time logics have been considered in [19].

**Frequency logics.** A major subject of our investigation are frequency constraints. This concept embeds naturally into the context of counting logics as it can be understood as a restricted form of counting. We therefore define in the following the frequency temporal logics $\text{fCTL}^*$, $\text{fLTL}$ and $\text{fCTT}$ as fragments of $\text{CCTL}^*$. Consider the following grammar defining the syntax of formulae $\varphi$ for natural numbers $n, m \in N$ with $n \leq m > 0$ and $p \in AP$.

\[
\varphi ::= p \mid \varphi \land \varphi \mid \neg \varphi \mid \alpha \\
\beta ::= X \varphi \mid \varphi U^\omega \varphi
\]

With the additional rule $\alpha ::= E \beta$ it defines precisely the set of $\text{fCTL}^*$ formulae while it defines $\text{fCTL}$ for $\alpha ::= E \beta \mid A \beta$ and $\text{fLTL}$ for $\alpha ::= \beta$. The semantics is defined by interpreting $\text{fCTL}^*$ formulae as $\text{CCTL}^*$ with the additional equivalence

\[
\varphi U^\omega \psi \overset{\text{def}}{=} \psi \lor x. F ((X \psi) \land m \cdot \#_x(\varphi) \geq n \cdot \#_x(T))
\]

for $\text{fCTL}^*$ formulae $\varphi$ and $\psi$ and a variable $x \in V$ not being used in either $\varphi$ or $\psi$.

**Example 1.** Consider the Kripke structure given by Figure 1 and the $\text{CTL}$ formula $\varphi_1 = x. A G (\neg (\#_2(p) \leq \#_1(EX r))).$ It basically states that on every path reaching $s_5$ there must be a position where the states $s_4$ and $s_4$ (satisfying $EX r$) together have been visited at least as often as the state $s_0$. A different, yet similar statement can be formulated using only frequency constraints: $\varphi'_1 = \#_1(\#_1entials X r) \geq 5 q$ states that $s_5$ must always be reached while visiting
Figure 1 A flat Kripke over $AP = \{p, q, r\}$.

$s_2$ and $s_4$ together at least as often as $s_0$, $s_1$ and $s_3$. Both $\varphi_1$ and $\varphi'_1$ are violated, e.g. by the path $s_0s_1s_2s_4s_5$. The Kripke structure however satisfies $\varphi_2 = z. A G (\neg q \rightarrow E F z(p) < z(r))$ because from every state except $s_5$ the number of positions that satisfy $r$ can be increased arbitrarily without increasing the number of those satisfying $p$. Notice that this would not be the case, e.g., if $s_4$ was labelled by $p$.

While the positional variables in $CCTL^*$ are a very flexible way of defining the scope of a constraint, frequency constraints in $fCTL^*$ are always bound to the scope of an until operator. The same applies to the counting constraints of $cLTL$ as defined in [19]. For example, the $cLTL$ formula $\varphi U[\sum_{i=1}^{n} a_i \#(\varphi_i) + a_n \#(\varphi_n) \geq k] \psi$ is equivalent to the $CLTL$ formula $z. \varphi (\psi \land a_1 \#z(\varphi_1) + \cdots + a_n \#z(\varphi_n) \geq k)$. Admitting only natural coefficients, $cLTL$ can be encoded even in $LTL$ making it thus strictly less expressive than $fLTL$. On the other hand, $cLTL_\pm$ admits arbitrary integer coefficients, which is more general than the frequency until operator of $fLTL$. For example, $p U a b q$ can be expressed as $\top U[\sum_{i=1}^{n} a_i \#(\top) - a_i \#(\top) \geq 0] q$ in $cLTL_\pm$.

Model-checking problem. We now present the problem on which we focus our attention. The model-checking problem for a class $\mathcal{K} \subseteq \mathcal{KS}$ of Kripke structures and a specification language $L$ (in our case all the specification languages are fragments of $CCTL^*$) is denoted by $MC(\mathcal{K}, L)$ and defined as the following decision problem.

**Input:** A Kripke structure $K \in \mathcal{K}$ and a formula $\varphi \in L$.

**Decide:** Does $K \models \varphi$ hold?

For temporal logics without counting variables, the model-checking problem over Kripke structure has been studied intensively and is known to be PSPACE-complete for LTL and CTL* and in P for CTL (see e.g. [3]). It has recently been shown that when restricting to flat (or weak) structures the complexity of the model-checking problem for LTL is lower than in the general case [17]: it drops from PSPACE to NP. As we show later, in the case of $CCTL^*$, flatness of the structures allows us to regain decidability of the model-checking problem which is in general undecidable. In this paper, we propose various ways to solve the model-checking problem of fragments of $CCTL^*$ over flat structures. For some of them we provide a direct algorithm, for others we reduce our problem to the satisfiability problem of a decidable extension of Presburger arithmetic.

3 Model-checking Frequency CTL

Satisfiability of $fLTL$ is undecidable [5] implying the same for model-checking $fLTL$, $CLTL$ and $CCTL^*$ over Kripke structures. This applies moreover to $CCTL$ [19]. In contrast, we show in the following that $MC(\mathcal{KS}, fCTL)$ is decidable using an extension of the well-known labelling algorithm for $CTL$ (see e.g. [3]).
Let $K = (S, s_1, E, \lambda)$ be a Kripke structure and $\Phi$ an $\mathcal{fCTL}$ formula. We compute recursively subsets $S_\varphi \subseteq S$ of the states of $K$ for every subformula $\varphi \in \text{sub}(\Phi)$ of $\Phi$ such that for all $s \in S$ we have $s \in S_\varphi$ iff $s \models \varphi$. Checking whether the initial state $s_1$ is contained in $S_\varphi$ then solves the problem. Propositions ($p \in AP$), negation ($\neg \varphi$), conjunction ($\varphi \land \psi$) and temporal next ($\mathcal{E} \varphi$, $\mathcal{A} \varphi$) are handled as usual, e.g., $S_p = \{ q \in S \mid p \in \lambda(q) \}$ and $S_{\mathcal{E} \varphi} = \{ q \in S \mid \exists q' \in S_\varphi : (q, q') \in \delta \}$.

To compute if a state $s \in S$ satisfies a formula of the form $\mathcal{E} \varphi \lor \psi$ or $\mathcal{A} \varphi \lor \psi$, assume that $S_\varphi$ and $S_\psi$ are given inductively. If $s \in S_\psi$ we immediately have $s \in S_{\mathcal{E} \varphi \lor \psi}$ and $s \in S_{\mathcal{A} \varphi \lor \psi}$. For the remaining cases, the problem of deciding whether $s \in S_{\mathcal{E} \varphi \lor \psi}$ or $s \in S_{\mathcal{A} \varphi \lor \psi}$, respectively, can be reduced in linear time to the repeated control-state reachability problem in systems with one integer counter. The idea is to count the ratio along paths $\rho \in S^\omega$ in $K$ as follows, in direct analogy to the semantics defined in Equation 1. Assume $r = \frac{n}{m}$ for $n, m \in \mathbb{N}$ and $n \leq m$. For passing any position on $\rho$ we pay a fee of $n$ and for those positions that satisfy $\varphi$ we gain a reward of $m$. Thus, we obtain a non-negative balance of rewards and gains at some position on $\rho$ if, in average, among every $m$ positions there are at least $n$ positions that satisfy $\varphi$, meaning the ratio constraint is satisfied. In $K$, this balance along a path can be tracked using an integer counter that is increased by $m - n$ when leaving a state $s' \in S_\varphi$ and decreased by adding $-n$ whenever leaving a state $s' \notin S_\varphi$. Thus, let $\hat{K}_s = (S, s, \{c\}, \Delta)$ be the counter system with

$$\Delta = \{(t, u, 0, t') \mid (t, t') \in E, t \not\in S_\varphi \Rightarrow u(c) = -n, t \in S_\varphi \Rightarrow u(c) = m - n\}.$$ 

The state $s$ satisfies the formula $\mathcal{A} \varphi \lor \psi$ if there is no path starting in state $s$ violating the formula $\varphi \lor \psi$. The latter is the case if at every position where $\psi$ holds, the balance computed up to this position is negative. Therefore, consider an extension $R_s$ of $\hat{K}_s$ where every edge leading into a state $s' \in S_\psi$ is guarded by the constraint $c < 0$. Every (infinite) run of $R_s$ is now a counter example for the property holding at $s$. To decide whether $s \in S_{\mathcal{A} \varphi \lor \psi}$ it suffices to check that in $R_s$ no state is repeatedly reachable from $s$.

A formula $\mathcal{E} \varphi \lor \psi$ is satisfied by $s$ if there is some state $s' \in S_\varphi$ reachable from $s$ with a non-negative balance. Hence, consider the counter system $U_s = (S \cup \{t\}, s, \{c\}, \Delta')$ obtained from $\hat{K}_s$ featuring a new sink state $t \not\in S$. The transition relation

$$\Delta' = \Delta \cup \{(s', 0, \{c \geq 0\}, t) \mid s' \in S_\varphi \} \cup \{(t, 0, 0, t)\}$$

extends $\Delta$ such that precisely the paths starting in $s$ and reaching a state $s' \in S_\varphi$ with non-negative counter value (i.e. sufficient ratio) can be extended to reach $t$. Checking if $s$ is supposed to be contained in $S_{\mathcal{E} \varphi \lor \psi}$ then amounts to decide whether $t$ is (repeatedly) reachable from $s$ in $U_s$.

Finally, repeated reachability is easily translated to the accepting run problem of B"uchi pushdown systems (BPDS) and the latter is in P [6]. A counter value $n \geq 0$ can be encoded into a stack of the form $\oplus^n$ while $\ominus^n$ encodes $n \leq 0$ and for evaluating the guards $c \geq 0$ and $c < 0$ only the top symbol is relevant. Simulating an update of the counter by a number $a \in \mathbb{Z}$ requires to perform $|a|$ push or pop actions. The size of the system is therefore linear in the largest absolute update value and hence exponential in its binary representation. Since the updates of the constructed counter systems originate from the ratios in $\Phi$, the corresponding BPDS are of up to exponential size in $|\Phi|$. During the labelling procedure this step must be performed at most a polynomial number of times giving an exponential-time algorithm.

\begin{itemize}
  \item \textbf{Theorem 2.} $MC(KS, \text{\mathcal{fCTL}})$ is in Exp.
\end{itemize}
It is worth noting that for a fixed formula (program complexity) or a unary encoding of numbers in frequency constraints, the size of the constructed Büchi pushdown systems and thus the runtime of the algorithm remains polynomial.

\begin{corollary}
MC(\text{KS}, f\text{CTL}) with unary number encoding is in \text{P}.
\end{corollary}

\section{Model-checking Frequency LTL over Flat Kripke Structures}

We show in this section that model-checking \text{fLTL} is decidable over flat Kripke structures. As decision procedure we employ a guess and check approach: given a flat Kripke structure \( \mathcal{K} \) and an \text{fLTL} formula \( \Phi \), we choose non-deterministically a set of satisfying runs to witness \( \mathcal{K} \models \Phi \). As representation for such sets we introduce augmented path schemas that extend the concept of path schemas \cite{20, 11} and provide for each of its runs a labelling by formulae. We show that if an augmented path schema features a syntactic property that we call consistency then the associated runs actually satisfy the formulae they are labelled with. Moreover, we show that every run of \( \mathcal{K} \) is in fact represented by some consistent schema of size at most exponential in \(|\mathcal{K}| + |\Phi|\). This gives rise to the following non-deterministic procedure.

1. \textbf{Read as input} an FKS \( \mathcal{K} \) and an \text{fLTL} formula \( \Phi \).
2. \textbf{Guess} an augmented path schema \( \mathcal{P} \) in \( \mathcal{K} \) of at most exponential size.
3. \textbf{Terminate} successfully if \( \mathcal{P} \) is consistent and accepts a run that is initially labelled by \( \Phi \).

We fix for this section a flat Kripke structure \( \mathcal{K} = (S, s_1, E, \lambda) \) and an \text{fLTL} formula \( \Phi \). For convenience we assume that \( AP \subseteq \text{sub}(\Phi) \). Omitted technical details can be found in \cite{9}.

\subsection{Augmented Path Schemas}

The set of runs of \( \mathcal{K} \) can be represented as a finite number of so-called path schemas that consist of a sequence of paths and simple loops consecutive in \( \mathcal{K} \) \cite{20, 11}. A path schema represents all runs that follow the given shape while repeating each loop arbitrarily often. For our purposes we extend this idea with additional labellings and introduce integer counters, updates and guards that can restrict the admitted runs.

\begin{definition}[Augmented Path Schema]
An augmented state of \( \mathcal{K} \) is a tuple \( a = (s, L, G, u, t) \in S \times 2^{\text{sub}(\Phi)} \times 2^{\text{st}(C)} \times \mathbb{Z}^C \times \{L, R\} \) comprised of a state \( s \) of \( \mathcal{K} \), a set of formula labels \( L \), guards \( G \) and an update \( u \) over a set of counter names \( C \), and a type \( t \) indicating whether the state is part of a loop (L) or a not (R). We denote by \( \text{st}(a) = s \), \( \lambda \text{ab}(a) = L \), \( g(a) = G \), \( u(a) = u \) and \( t(a) = t \) the respective components of \( a \). An augmented path in \( \mathcal{K} \) is a sequence \( u = a_0 \ldots a_n \) of augmented states \( a_i \) such that \( (\text{st}(a_i), \text{st}(a_{i+1})) \in E \) for \( i \in [0, n - 1] \). If \( t(a_i) = R \) for all \( i \in [0, n - 1] \) then \( u \) is called a row. It is called an augmented simple loop (or simply loop) if it is non-empty and \( (\text{st}(a_0), \text{st}(a_1)) \in E \) and \( \text{st}(a_i) \neq \text{st}(a_j) \) for \( i \neq j \) and \( t(a_i) = L \) for all \( i \in [0, n - 1] \).

An augmented path schema (APS) in \( \mathcal{K} \) is a tuple \( \mathcal{P} = (P_0, \ldots, P_n) \) where each component \( P_k \) is a row or a loop, \( P_n \) is a loop and their concatenation \( P_0 P_1 P_2 \ldots P_n \) is an augmented path.

Thanks to counters we can, for example, restrict to those runs satisfying a specific frequency constraint at some positions tracking it as discussed in Section 3. Figure 2 shows an example of an APS with edges indicating the possible state progressions. It features a single counter that tracks the frequency constraint of a formula \( rU^2 q \) from state 1.

We denote by \(|\mathcal{P}| = |P_0 \ldots P_n|\) the size of \( \mathcal{P} \) and use global indices \( \ell \in [0, |\mathcal{P}| - 1] \) to address the \((\ell + 1)\)-th augmented state in \( P_0 \ldots P_n \), denoted \( \mathcal{P}[\ell] \). To distinguish these global indices from positions in arbitrary sequences, we refer to them as locations of \( \mathcal{P} \). Moreover,
We are interested in this labelling being \( \theta \Psi = APS \) is called An APS \( P_0 \) of the flat Kripke structure in Figure 1.

\[
\text{Figure 2} \quad \text{An APS } P = (P_0, \ldots, P_7) \text{ of the flat Kripke structure in Figure 1.}
\]

\( \text{loc}_P(k) = \{ \ell \mid |P_0 \ldots P_{k-1}| \leq \ell < |P_0 \ldots P_k| \} \) denotes for \( 0 \leq k \leq n \) the set of locations belonging to component \( P_k \) and for all locations \( \ell \in \text{loc}_P(k) \) we denote the corresponding component index in \( P \) by \( \text{comp}_P(\ell) = k \). For example, in Figure 2 we have \( \text{loc}_P(3) = \{3, 4\} \) and \( \text{comp}_P(6) = 5 \) because the seventh state of \( P_0 \) belongs to \( P_5 \). We extend the component projections for augmented states to (sequences of) locations of \( P \) and write, e.g., \( \text{st}_P(\ell_1 \ell_2) \) for \( \text{st}(P[\ell_1]) \text{st}(P[\ell_2]) \) and \( u_P(\ell) \) for \( u(P[\ell]) \).

An APS \( P \) gives rise to a counter system \( \text{CS}(P) = (Q, 0, C, \Delta) \) where \( Q = \{0, \ldots, |P| - 1\} \), \( C \) are the counters used in the augmented states of \( P \) and \( \Delta \) consists of those transitions \( (\ell, u_P(\ell), g_P(\ell'), \ell') \) such that \( 0 \leq \ell' = \ell + 1 < |P| \) or \( \ell' < \ell \) and \( \{\ell, \ell + 1, \ldots, \ell'\} = \text{loc}_P(k) \) for some loop \( P_k \). Notice that the APS in Figure 2 is presented as its corresponding counter system. Let \( \text{succ}_P(\ell) \) denote the set \( \{\ell' \in Q \mid \exists u, G : (\ell, u, G, \ell') \in \Delta\} \) of successors of \( \ell \) in \( \text{CS}(P) \). A run of \( P \) is a run of \( \text{CS}(P) \) that visits each location \( \ell \in S \) at least once. The set of all runs of \( P \) is denoted \( \text{Runs}(P) \). As a consequence, a run visits the last loop infinitely often. We say that an APS \( P \) is non-empty iff \( \text{Runs}(P) \neq \emptyset \). Since every run \( \sigma \in \text{Runs}(P) \) corresponds, by construction of \( P \), to a path \( \text{st}_P(\rho) \in Q^\omega \) in \( K \) we define the satisfaction of an \( \text{fLTL} \) formula \( \varphi \) at position \( i \) by \( (\sigma, i) \models_\varphi \text{ if } (\text{st}_P(\sigma), i) \models \varphi \).

Finally, notice that \( \text{CS}(P) \) is in fact a flat counter system. It is shown in [11] that LTL properties can be verified over flat counter systems in non-deterministic polynomial time. Since LTL can express that each location of \( \text{CS}(P) \) is visited we obtain the following result.

\[\text{Lemma 5} \text{[11]. Deciding non-emptiness of APS is in NP.}\]

### 4.2 Labellings of Consistent APSs are Correct

An APS \( P \) assigns to every position \( i \) on each of its runs \( \sigma \) the labelling \( L_i = \text{lab}_P(\sigma(i)) \).

We are interested in this labelling being \textit{correct} with respect to some \( \text{fLTL} \) formula \( \Phi \) in the sense that \( \Phi \in L_i \) if and only if \( (\sigma, i) \models_\Phi \). The notion of consistency introduced in the following provides a sufficient criterion for correctness of the labelling of all runs of an APS.

An augmented path \( u = a_0 \ldots a_n \) is said to be \textit{good}, \textit{neutral} or \textit{bad} for an \( \text{fLTL} \) formula \( \Psi = \varphi \U^{\approx d} \psi \) if the number \( d = |\{0 \leq i < |u| \mid \varphi \in \text{lab}(u(i))\}| \) of positions labelled with \( \varphi \) is larger than \( (d > \frac{x}{y} \cdot |u|) \), equal to \( (d = \frac{x}{y} \cdot |u|) \) or smaller than \( (d < \frac{x}{y} \cdot |u|) \), respectively, the fraction \( \frac{x}{y} \) of all positions of \( u \). A tuple \( (P_0, \ldots, P_n) \) of rows and loops (not necessarily an APS) is called \textit{L-periodic} for a set \( L \subseteq \text{sub}(\Phi) \) of labels if all augmented paths \( P_k \) share the same labelling with respect to \( L \), that is for all \( 0 \leq k < n - 1 \) we have \( |P_k| = |P_{k+1}| \) and \( \text{lab}(P_k(i)) \cap L = \text{lab}(P_{k+1}(i)) \cap L \) for all \( 0 \leq i < |P_k| \).

\[\text{Definition 6} \text{[Consistency]. Let } P = (P_0, \ldots, P_n) \text{ be an APS in } K, k \in [0, n] \text{ and } \ell \in \text{loc}_P(k) \text{ a location on component } P_k. \text{ The location } \ell \text{ is } \textit{consistent} \text{ with respect to an } \text{fLTL} \text{ formula } \Psi \text{ if all locations of } P \text{ are consistent with respect to all strict subformulae of } \Psi \text{ and one of the following conditions applies.}\]

\[\boxed{CONCUR 2017}\]
1. \( \Psi \in AP \) and \( \Psi \in \text{lab}_P(\ell) \iff \Psi = \lambda(\text{st}_P(\ell)) \), or \( \Psi = \varphi \land \psi \) and \( \Psi \in \text{lab}_P(\ell) \iff \varphi, \psi \in \text{lab}_P(\ell) \), or \( \Psi = \neg \varphi \) and \( \Psi \in \text{lab}_P(\ell) \iff \varphi \not\in \text{lab}_P(\ell) \).

2. \( \Psi = X \varphi \) and \( \forall \ell' \in \text{succ}_P(\ell) : \Psi \in \text{lab}_P(\ell) \iff \varphi \in \text{lab}_P(\ell') \).

3. \( \Psi = \varphi \lor \psi \) and one of the following holds:
   a. \( \Psi, \psi \in \text{lab}_P(\ell) \)
   b. \( \Psi \in \text{lab}_P(\ell) \) and \( P_n \) is good for \( \Psi \) and \( \exists \ell' \in \text{loc}_P(n) : \psi \in \text{lab}_P(\ell') \)
   c. \( \text{t}_P(\ell) = R \) and there is a counter \( c \in C \) such that \( \forall \ell' < \ell : \text{u}_P(\ell')(c) = 0 \) and \( \forall \ell' \geq \ell : \varphi \in \text{lab}_P(\ell') \Rightarrow \text{u}_P(\ell')(c) = y - x \) and \( \forall \ell' \geq \ell : \psi \not\in \text{lab}_P(\ell') \) and \( \psi \in \text{lab}_P(\ell') \Rightarrow (c < 0) \in \text{g}_P(\ell') \)
   d. There is \( k' \in [0, n] \) such that all locations \( \ell' \in \text{loc}_P(k') \) are consistent wrt. \( \Psi \) and
      - if \( k = n \) then \( k' < k \) and \( (P_{k'}, P_{k'+1}, \ldots, P_k) \) is \( \{\varphi, \psi, \Psi\}\)-periodic,
      - if \( k < n \) and \( P_k \) is good or neutral for \( \Psi \) and \( \Psi \not\in \text{lab}_P(\ell) \), or \( P_k \) is bad for \( \Psi \) and \( \Psi \in \text{lab}_P(\ell) \) then \( k' < k < n \) and \( (P_{k'}, P_{k'+1}, \ldots, P_{k+1}) \) is \( \{\varphi, \psi, \Psi\}\)-periodic, and
      - if \( k < n \) and \( P_k \) is good or neutral for \( \Psi \) and \( \Psi \in \text{lab}_P(\ell) \), or \( P_k \) is bad for \( \Psi \) and \( \Psi \not\in \text{lab}_P(\ell) \) then \( k < k' < n \) and \( (P_{k'}, P_{k'+1}, \ldots, P_{k+1}) \) is \( \{\varphi, \psi, \Psi\}\)-periodic.

The APS \( P \) is consistent with respect to \( \Psi \) if it is the case for all its locations.

The cases 1 and 2 reflect the semantics syntactically. For instance, location 0 in Figure 2 can be labelled consistently with \( X p \) since all its successor (0 and 1) are labelled with \( p \).

Case 3, concerning the (frequency) until operator, is more involved.

Assume that \( \Phi = \varphi \lor \psi \) is an until formula and that the labelling of \( K \) by \( \varphi \) and \( \psi \) is consistent. In some cases, it is obvious that \( \Phi \) holds, namely at positions labelled by \( \psi \) (case 3a) or if the final loop already guarantees that \( \Phi \) always holds (case 3b). If neither is the case we can apply the idea discussed in Section 3 and use a counter to check explicitly if at some point the formula \( \Phi \) holds (case 3c). Recall that to validate (or invalidate) the labelling of a location by the formula \( \Phi \) a specific counter tracks the frequency constraint in terms of the balance between fees and rewards along a run. For the starting point to be unique this case only applies to locations that are not part of a loop. For those labelled with \( \Phi \) there should exist a location in the future where \( \psi \) holds and the balance counter is non-negative. For those not labelled with \( \Phi \) all locations in the future where \( \psi \) holds must be entered with negative balance. Finally, case 3d can apply (not only) to loops and is based on the following reasoning: if a loop is good (bad) and \( \Phi \) is supposed to hold at some of its locations then it suffices to verify that this is the case during any of its future (past) iterations, e.g. the last (first) and vice versa if \( \Phi \) is supposed not to hold. This is the reason why this case allows for delegating consistency along a periodic pattern.

For instance, consider the formula \( \Psi = r \lor \Psi \) and the APS shown in Figure 2. It is consistent to not label location 1 by \( \Psi \) because the counter \( c \) tracks the balance and locations 7 and 8 are guarded as required. If a run takes, e.g., the loop \( P_0 \) seven times, it has to take \( P_1 \) at least twice to satisfy all guards. This ensures that the ratio for the proposition \( r \) is strictly less than \( \frac{1}{2} \) upon reaching the first (and thus any) occurrence of \( q \). Note that to also make location 2 consistent, an additional counter needs to be added. Consistency with respect to \( \Psi \) is then inherited by location 0 from location 1 according to case 3d of the definition. Intuitively, additional iterations of the bad loop \( P_0 \) can only diminish the ratio.

The definition of consistency guarantees that if an APS is consistent with respect to \( \Phi \) then for every run of the APS, each time the formula \( \Phi \) is encountered, it holds at the current position (see [9] for further details). Hence we obtain the following lemma that guarantees correctness of our decision procedure.
Lemma 7 (Correctness). If there is an APS $\mathcal{P}$ in $\mathcal{K}$ such that $\mathcal{P}$ is consistent wrt. $\Phi$ and $\Phi \in \text{lab}_0(\mathcal{P})$ and $\text{Runs}(\mathcal{P}) \neq \emptyset$ then $\mathcal{K} \models \Phi$.

4.3 Constructing Consistent APS

Assuming that our flat Kripke structure $\mathcal{K}$ admits a run $\rho$ such that $\rho \models \Phi$, we show how to construct a non-empty APS that is initially labelled by and consistent with respect to $\Phi$. It will be of at most exponential size in $|\mathcal{K}| + |\Phi|$ and is built recursively over the structure of $\Phi$.

Concerning the base case where $\Phi \in AP$, all paths in a flat structure can be represented by a path schema of linear size [20, 11]. Intuitively, since $\mathcal{K}$ is flat, every subpath $s_is_{i+1}\ldots s_{i'}\ldots s_{i''}$ of $\rho$ where a state $s_i = s_{i'} = s_{i''}$ occurs more than twice is equal to $(s_is_{i+1}\ldots s_{i'-1})^ks_{i''}$ for some $k \in \mathbb{N}$. Hence, there are simple subpaths $u_0, \ldots, u_m \in \mathbb{S}^+$ of $\rho$ and positive numbers of iterations $n_0, \ldots, n_{m-1} \in \mathbb{N}$ such that $\rho = u_0^n u_1^{n_1} \ldots u_{m-1}^{n_{m-1}} u_m^{|u_m|}$ and $|u_0u_1\ldots u_m| \leq 2|S|$. From this decomposition, we build an APS being consistent with respect to all propositions. Henceforth, we assume by induction an APS $\mathcal{P}$ being consistent with respect to all strict subformulae of $\Phi$ and a run $\sigma \in \text{Runs}(\mathcal{P})$ with $\text{st}_\mathcal{P}(\sigma) = \rho$. If $\Phi = \varphi \land \psi$ or $\Phi = \neg \varphi$, Definition 6 determines for each augmented state of $\mathcal{P}$ whether it is supposed to be labelled by $\Phi$ or not. It remains hence to deal with the next and frequency until operators.

Labelling $\mathcal{P}$ by $\chi \varphi$. If $\Phi = \chi \varphi$ the labelling at some location $\ell$ is extended according to the labelling of its successors. These may disagree upon $\varphi$ (only) if $\ell$ has more than one successor, i.e., being the last location on a loop $P_k$ of $\mathcal{P} = (P_0, \ldots, P_m)$. In that case we consult the run $\sigma$: if it takes $P_k$ only once, this loop can be cut and replaced by $P_k'$ that we define to be an exact copy except that all augmented states have type $\Phi$.

Labelling $\mathcal{P}$ by $\varphi \lor^* \psi$. The most involved case is to label a location $\ell$ by $\Phi = \varphi \lor^* \psi$. First, assume that $\ell$ is part of a row. Whether it must be labelled by $\Phi$ is uniquely determined by $\sigma$. This is consistent if case 3a or 3b of Definition 6 applies. The conditions of case 3c are also realised easily in most situations. Only, if $\Phi$ holds at $\ell$ but every location $\ell'$ witnessing this (by being reachable with sufficient frequency and labelled by $\psi$) is part of some loop $P'$. Adding the required guard directly to $\ell'$ may be too strict if $\sigma$ traverses $P'$ more than once. However, the first iteration (if $P'$ is bad for $\Phi$) or the last iteration (if $P'$ is good) on $\sigma$ contains a position (labelled with $\psi$) witnessing that $\Phi$ holds if any iteration does. Thus it suffices to unfold the loop once in the respective direction. For example, consider in Figure 2 location 5 and a formula $\varphi = r \lor^* q$. Location 8 could witness that $\varphi$ holds but a corresponding guard would be violated eventually since $P_7$ is bad for $\varphi$. The first iteration is thus the optimal choice. The unfolding $P_8$ separates it such that location 7 can be guarded instead without imposing unnecessary constraints.

Now assume that location $\ell$, to be labelled or not with $\Phi$, is part of a loop $P$ which is stable in the sense that $\Phi$ holds either at all positions $i$ with $\sigma(i) = \ell$ or at none of them. With two unfoldings of $P$, made consistent as above, case 3d applies. However, $\sigma$ may go through $\ell$ several, say $n > 1$, times where $\Phi$ holds at some but not all of the corresponding positions. If $n$ is small we can replace $P$ by precisely $n$ unfoldings, thus reducing to the
previous case without increasing the size of the structure too much. We can moreover show that if \( n \) is not small then it is possible to decompose such a problematic loop into a constant number of unfoldings and two stable copies based on the following observation.

**Lemma 8 (Decomposition).** Let \( P = P[\ell_0] \ldots P[\ell_{|P|-1}] \) be a non-terminal loop in \( P \) with corresponding location sequence \( v = \ell_0 \ldots \ell_{|P|-1} \) and \( \hat{n} = |P| \cdot y \) for some \( y > 0 \). For every run \( \sigma = uv^n w \in \text{Runs}(P) \) where \( n \geq \hat{n} + 2 \) there are \( n_1 \) and \( n_2 \) such that \( \sigma = uv^{n_1} v^h v^{n_2} w \) and for all positions \( i \) on \( \sigma \) with \( |u| \leq i < |uv^{n_1} w| \) or \( |uv^{n_1} v^h| \leq i < |uv^{n_1} v^{n_2} w| \) we have \( (\sigma,i) \models P \) if \( (\sigma,i+|P|) \models E \).

**Example 9.** Consider again the APS \( P \) in Figure 2, a run \( \sigma \in \text{Runs}(P) \) and the location 3. Whether or not \( \varphi = r \cup \exists q \) holds at some position \( i \) with \( \sigma(i) = 3 \) depends on how often \( \sigma \) traverses the good loop \( P_3 \) (the more the better) and how often it repeats \( P_3 \) after position \( i \) (the more the worse). Assume \( \sigma \) traverses \( P_3 \) exactly five times and \( P_3 \) sufficiently often, say 10 times. Then, during the last three iterations of \( P_3 \), \( \varphi \) holds when visiting location 3, and also location 4. In the two iterations before, the formula holds exclusively at location 4 and in any preceding iteration, it does not hold at all. Thus any labelling of \( P_3 \) would necessarily be incorrect. However, we can replace \( P_3 \) by four copies of it that are labelled as indicated in Figure 3 and \( \sigma \) can easily be mapped onto this modified structure.

The presented procedure for constructing an APS from the run \( \rho \) in \( K \) performs only linearly many steps in \( |F| \), namely one step for each subformula. It starts with a structure of size at most \( 2|K| \) and all modifications required to label an APS increase its size by a constant factor. Hence, we obtain an APS \( P_\Phi \) of size at most exponential in the length of \( \Phi \) and polynomial in the number of states of \( K \). This consistent APS still contains a run corresponding to \( \rho \) and hence its first location must be labelled by \( \Phi \) because \( (\rho, 0) \models \Phi \) and we have seen that consistency implies correctness.

**Lemma 10 (Completeness).** If \( K \models \Phi \) then there is a consistent APS \( P \) in \( K \) of at most exponential size in \( K \) and \( \Phi \) where \( \Phi \in \text{lab}(P(0)) \) and \( P \) is non-empty.

We have seen in this section that the decision procedure presented in the beginning is sound and complete due to Lemma 7 and 10, respectively. The guessed APS is of exponential size in \( |F| \) and of polynomial size in \( |K| \). Since both checking consistency and non-emptiness (cf. Lemma 5) require polynomial time (in the size of the APS) the procedure requires at most exponential time.

**Theorem 11.** \( \text{MC}(\text{FKS}, f\text{LTL}) \) is in \( \text{NEXP} \).

This result immediately extends to \( f\text{CTL}^* \). For a state \( q \) of a flat Kripke structure \( K \) and an arbitrary \( f\text{LTL} \) formula \( \varphi \), the procedure allows us to decide in \( \text{NEXP} \) whether \( q \models E \varphi \).
holds. It allows us further to decide if \( q \models \varphi \) holds in \( \text{ExpSpace} \) by the dual formulation \( q \not\equiv \exists x \varphi \) and Savitch’s theorem. Following otherwise the standard labeling procedure for \( \text{CTL} \)

(cf. Section 3) requires to invoke the procedure a polynomial number of times in \(|K| + |\Phi|\).

\[ \text{Theorem 12.} \quad \text{MC}(\text{FKS, fCTL}^*) \text{ is in } \text{ExpSpace}. \]

5 On model-checking \( \text{CCTL}^* \) over flat Kripke structures

In this section, we prove decidability of \( \text{MC}(\text{FKS, CCTL}^*) \). We provide a polynomial encoding into the satisfiability problem of a decidable extension of Presburger arithmetic featuring a quantifier for counting the solutions of a formula. For the reverse direction an exponential reduction provides a corresponding hardness result for \( \text{CLTL, CCTL} \) and \( \text{CCTL}^* \).

Presburger arithmetic with Härtig quantifier. First-order logic over the natural numbers with addition was shown to be decidable by M. Presburger [23]. It has been extended with the so-called Härtig quantifier [2, 24, 25] that allows for referring to the number of values for a specific variable that satisfy a formula. We denote this extension by \( \text{PH} \). The syntax of \( \text{PH} \) formulae \( \varphi \) and \( \text{PH} \) terms \( \tau \) over a set of variables \( V \) is defined by the grammar

\[
\varphi ::= \tau \leq \tau | \neg \varphi | \varphi \land \varphi | \exists x. \varphi | \exists^= x.y. \varphi
\]

\[
\tau ::= a | a \cdot x | \tau + \tau
\]

for natural constants \( a \in \mathbb{N} \) and variables \( x, y \in V \). Since the structure \( (\mathbb{N},+) \) is fixed, the semantics is defined over valuations \( \eta : \mathbb{K} \to \mathbb{N} \) that are extended to terms \( t \) as expected, e.g.,

\[
\eta(3 \cdot x + 1) = 3 \cdot \eta(x) + 1.
\]

We define the satisfaction relation \( \models_{\text{PH}} \) as usual for first-order logics and by \( \eta = \models_{\text{PH}} \exists^= x.y. \varphi \) \( \mathbb{N} \ni \{ b \in \mathbb{N} | \eta[y \mapsto b] = \models_{\text{PH}} \varphi \} = \eta(x) \) for the Härtig quantifier. Notice that the solution set has to be finite.

The satisfiability problem of \( \text{PH} \) consists in determining whether for a \( \text{PH} \) formula \( \varphi \) there exists a valuation \( \eta \) such that \( \eta = \models_{\text{PH}} \varphi \). It is decidable [2, 24, 25] via eliminating the Härtig quantifier, but its complexity is not known. For what concerns classic Presburger arithmetic, the complexity of its satisfiability problem lies between \( 2\text{Exp} \) and \( 2\text{ExpSpace} \) [4].

Lower bound for \( \text{MC}(\text{FKS, CCTL}^*) \). Let \( K \) be the flat Kripke structure over \( AP = \emptyset \) that consists of a single loop of length one. We can encode satisfiability of a \( \text{PH} \) formula \( \Phi \) into the question whether the (unique) run \( \rho \) of \( K \) satisfies a \( \text{CLTL} \) formula \( \Phi \). Assume without loss of generality that \( \Phi \) has no free variables. Let \( V_\Phi \) be the variables used in \( \Phi \) and \( z_1, z_2, \ldots \not\in V_\Phi \) additional variables. Recall that \( \rho \models \Phi \) if \( (\rho, \theta, 0) \models \Phi \) for some valuation \( \theta \) of the positional variables in \( \Phi \).

The idea is essentially to encode the value given to a variable \( x \in V_\Phi \) of \( \Phi \) into the distance between the positions assigned to two variables of \( \Phi \). Technically, a mapping \( Z \subseteq \mathbb{N}^{1^{\Phi}} \) associates with each variable \( x \in V_\Phi \) an index \( j = Z(x) \) and the constraints that \( \Phi \) imposes on \( x \) are translated to constraints on positional variables \( z_j \) and \( z_{j-1} \) (more precisely, the distance \( \theta(z_j) - \theta(z_{j-1}) \) between the assigned positions). The following transformation \( t : \text{PH} \times \mathbb{N}^{1^{\Phi}} \times \mathbb{N} \to \text{CLTL} \) constructs the \( \text{CLTL} \) formula from \( \Phi \). When a variable is encountered, the mapping \( Z \) is updated by assigning to it the next free index (third parameter). Let

\[
\begin{align*}
t(\varphi_1 \circ \varphi_2, Z, i) &= t(\varphi_1, Z, i) \circ t(\varphi_2, Z, i) \\
t(a \cdot x, Z, i) &= a \cdot \#_{x(i)}(\top) - a \cdot \#_{x(i)}(\top) \\
t(\exists x. \varphi, Z, i) &= F z_i.t(\varphi, Z[x \mapsto i], i + 1) \\
t(\exists^= x.y. \varphi, Z, i) &= F G \{ t(x, Z, i) = \#_{z_{i-1}}(z_i.\varphi, Z[y \mapsto i, i + 1]) \}
\end{align*}
\]

for \( x, y \in V_\Phi, a, i \in \mathbb{N} \) and \( \circ \in \{ \land, \leq, + \} \). Then, we obtain \( \Phi = z_0.t(\Phi, 1, 1) \), initialising \( Z \) and the first free index with 1. Notice that the translation of the Härtig quantifier instantiates
the scope effectively twice when substituting the equality and thus the size of $\hat{\Phi}$ may at worst double with each nesting. Finally, we can equivalently add path quantifiers to all temporal operators in $\hat{\Phi}$ and obtain, syntactically, a $\text{CTL}^*$ formula.

\textbf{Theorem 13.} The satisfiability problem of $\text{PH}$ is reducible in exponential time to both $\text{MC}(\text{FKS, CTL})$ and $\text{MC}(\text{FKS, CTL}^*)$.

\textbf{Deciding $\text{MC}(\text{FKS, CTL}^*)$.} We provide a polynomial reduction to the satisfiability problem of $\text{PH}$. Given a flat Kripke structure $\mathcal{K}$ we can represent each run $\rho$ by a fixed number of naturals. We use a predicate $\text{Conf}$ that allows for accessing the $i$-th state on $\rho$ given its encoding and a predicate $\text{Run}$ characterising all (encodings of) runs in $\text{Runs}(\mathcal{K})$. Such predicates were shown to be definable by Presburger arithmetic formulae of polynomial size and used to encode $\text{MC}(\text{FKS, CTL}^*)$ \cite{12, 10}. We adopt this idea for $\text{MC}(\text{FKS, CTL}^*)$ and $\text{PH}$. Let $\mathcal{K} = (S, s_I, E, \lambda)$ and assume $S \subseteq \mathbb{N}$ without loss of generality. For $N \in \mathbb{N}$ let $V_N = \{r_1, \ldots, r_N, i, s\}$ be a set of variables that we use to encode a run, a position and a state, respectively.

\textbf{Lemma 14 (\cite{10}).} There is a number $N \in \mathbb{N}$, a mapping $\text{enc} : \mathbb{N}^N \to S^N$ and predicates $\text{Conf}(r_1, \ldots, r_N, i, s)$ and $\text{Run}(r_1, \ldots, r_N)$ such that for all valuations $\eta : V_N \to \mathbb{N}$ we have

1. $\eta \models_{\text{PH}} \text{Run}(r_1, \ldots, r_N) \iff \text{enc}(\eta(r_1), \ldots, \eta(r_N)) \in \text{Runs}(\mathcal{K})$ and
2. if $\eta \models_{\text{PH}} \text{Run}(r_1, \ldots, r_N)$ then
   
   $\eta \models_{\text{PH}} \text{Conf}(r_1, \ldots, r_N, i, s) \iff \text{enc}(\eta(r_1), \ldots, \eta(r_N))(\eta(i)) = \eta(s)$.
   Both predicates are definable by $\text{PH}$ formulae over variables $V \supseteq V_N$ of polynomial size in $|\mathcal{K}|$.

Now, let $\Phi$ be a $\text{CTL}^*$ formula to be verified on $\mathcal{K}$. Without loss of generality we assume that all comparisons $\varphi \leq \varphi'$ of the form $r_1 \leq r_2$ have the shape $\varphi \leq \sum_{t=1}^{k} a_t \cdot \#x_t(\varphi_t) + b \leq \sum_{t=1}^{m} a_t \cdot \#x_t(\varphi_t) + c$ for some $k, m, b, c \in \mathbb{N}$, coefficients $a_t \in \mathbb{N}$ and subformulae $\varphi_t$. As it is done in \cite{10} for CTL, using the predicates $\text{Conf}$ and $\text{Run}$, we construct a $\text{PH}$ formula that is satisfiable if and only if $\mathcal{K} \models \Phi$. Given the encoding of relevant runs into natural numbers we can express path quantifiers with quantification over the variables $r_1, \ldots, r_N$. Temporal operators can be expressed by using $\text{Conf}$ to access specific positions. Storing of positions is done explicitly by assigning them as value to specific variables $x$. Variables $z$ are introduced to hold the number of positions satisfying a formula and can then be used in constraints. For example, to translate a term $\#x(\varphi)$ we specify a variable, e.g., $z_1$ holding this value by $\exists z_1. \exists z_2' i.x \leq i' \leq i \land \varphi$ where $i$ holds the current position and $\varphi$ expresses that $\varphi$ holds at position $i'$ of the current run. Constraints like $\#x(\varphi) + 1 \leq \#x(\psi)$ can now directly be translated to, e.g., $z_1 + 1 \leq z_2$. We use a syntactic translation function $\text{chk}$ that takes the formula $\varphi$ to be translated, the names of $N$ variables encoding the current run and the name of the variable holding the current position. Let

\[
\begin{align*}
\text{chk}(p, r_1, \ldots, r_N, i) &= \exists s. \text{Conf}(r_1, \ldots, r_N, i, s) \land \forall_{a \in A(a)} s = a \\
\text{chk}(\varphi \land \psi, r_1, \ldots, r_N, i) &= \text{chk}(\varphi, r_1, \ldots, r_N, i) \land \text{chk}(\psi, r_1, \ldots, r_N, i) \\
\text{chk}(\neg \varphi, r_1, \ldots, r_N, i) &= \neg \text{chk}(\varphi, r_1, \ldots, r_N, i) \\
\text{chk}(X \varphi, r_1, \ldots, r_N, i) &= \exists i'. i' = i + 1 \land \text{chk}(\varphi, r_1, \ldots, r_N, i') \\
\text{chk}(\varphi \lor \psi, r_1, \ldots, r_N, i) &= \exists i'. i \leq i' \land \text{chk}(\psi, r_1, \ldots, r_N, i') \land \\
&\quad \forall i'. (i \leq i' \land i' < i'' \land \text{chk}(\varphi, r_1, \ldots, r_N, i) \rightarrow \text{chk}(\psi, r_1, \ldots, r_N, i'')) \\
\text{chk}(\text{E} \varphi, r_1, \ldots, r_N, i) &= \exists r_1' \ldots \exists r_N'. \text{Run}(r_1', \ldots, r_N') \land \text{chk}(\varphi, r_1', \ldots, r_N', i) \land \forall i'. (i' \leq i) \rightarrow \exists s. \text{Conf}(r_1, \ldots, r_N, i', s) \land \text{Conf}(r_1', \ldots, r_N', i', s) \\
\text{chk}(x \varphi, r_1, \ldots, r_N, i) &= \exists x. x = i \land \text{chk}(\varphi, r_1, \ldots, r_N, i) \\
\text{chk}(\varphi \leq, r_1, \ldots, r_N, i) &= \exists z_1 \ldots \exists z_m. (\forall_{z_1} \exists z_2'. x_1 \leq i' \leq i \land \text{chk}(\varphi_1, r_1, \ldots, r_N, i') \land \\
&\quad a_1 \cdot z_1 + \ldots + a_k \cdot z_k + b \leq a_{k+1} \cdot z_{k+1} + \ldots + a_m \cdot z_m + c)
\end{align*}
\]
for $\varphi \triangleq \sum_{i=1}^{k}a_i \cdot \#_{x_i}(\varphi_i) + b \leq \sum_{i=k+1}^{m}a_i \cdot \#_{x_i}(\varphi_i) + c$. Primed variables denote fresh copies of the corresponding input variables, e.g. $i'$ becomes $(i')' = i''$ and $i''$ becomes $i'''$.

Now, $\Phi \models K$ if and only if $\exists r_1 \ldots \exists r_N. \exists i.Run(r_1, \ldots, r_N) \land i = 0 \land \text{chk}(\Phi, r_1, \ldots, r_N, i)$ is satisfiable.

Theorem 15. $\text{MC}(\text{FKS, CCTL}^*)$ is reducible to $\text{PH}$ satisfiability in polynomial time.

6 Conclusion

In this paper, we have seen that model checking flat Kripke structures with some expressive counting temporal logics is possible whereas this is not the case for general, finite Kripke structures. However, our results provide an under-approximation approach to this latter problem that consists in constructing flat sub-systems of the considered Kripke structure. We furthermore believe our method works as well for flat counter systems. We left as open problem the precise complexity for model checking $\text{fCTL}$, $\text{fLTL}$ and $\text{fCTL}^*$ over flat Kripke structures. It follows from [17] that the latter two problems are NP-hard while we obtain exponential upper bounds. However, we believe that if we fix the nesting depth of the frequency until operator in the logic, the complexity could be improved.

This work has shown, as one could have expected, a strong connection between $\text{CLTL}$ and counter systems and as future work we plan to study automata-based formalisms inspired by $\text{fLTL}$ where we will equip our automata with some counters whose role will be to evaluate the relative frequency of particular events.

References


29:16 Model-Checking Counting Temporal Logics on Flat Structures


Concurrent Reversible Sessions

Ilaria Castellani¹, Mariangiola Dezani-Ciancaglini¹², and Paola Giannini¹³

¹ Université Côte d’Azur, INRIA, 2004 Route des Lucioles, 06902 Sophia Antipolis, France ilaria.castellani@inria.fr
² Dipartimento di Informatica, Università di Torino, corso Svizzera 185, 10131 Torino, Italy dezani@di.unito.it
³ DiSIT, Università del Piemonte Orientale, Via Teresa Michel 11, 15121 Alessandria, Italy paola.giannini@uniupo.it

Abstract
We present a calculus for concurrent reversible multiparty sessions, which improves on recent proposals in several respects: it allows for concurrent and sequential composition within processes and types, it gives a compact representation of the past of processes and types, which facilitates the definition of rollback, and it implements a fine-tuned strategy for backward computation. We propose a refined session type system for our calculus and show that it enforces the expected properties of session fidelity, forward and backward progress, as well as causal consistency. In conclusion, our calculus is a conservative extension of previous proposals, offering enhanced expressive power and refined analysis techniques.

Keywords and phrases Communication-centric Systems, Reversible Computation, Process Calculi, Multiparty Session Types

1 Introduction
Building on recent proposals, we argue that session types and reversibility can be fruitfully combined to yield a model for correct and reliable communication-centric systems. Session types are a simple but expressive type formalism that specifies the structure of interactions. Traditionally, session types have been used to ensure safety properties of interactions, such as absence of communication errors, deadlock freedom and race freedom.

Reversibility, on its side, may be viewed as a means to improve system flexibility and reliability. Reversing a computation may be defined as the act of undoing some suffix of the computation, in order to return to a previously visited state. A condition which is usually required for undoing a computational step is that all its effects have been already undone [21, 22, 30, 33, 8, 16, 15]. This property is generally referred to as causal consistency.
A stronger property, called full reversibility in [25], enables a system to restore exactly a past state, keeping no trace of the rollback [9, 20, 24]. Sometimes neither of these properties can be achieved, particularly in distributed systems, and one has to go for weaker properties. In some cases, it is even desirable that a restored state be not exactly the same as the original one; for instance, the restored state could keep some memory of the undone computational path, so as to avoid engaging in that path again in case it led to an unsuccessful state.

In the setting of structured communications, reversibility has been first studied for contracts [2, 3] and transactions [10, 11, 19]. Only recently has this issue started to be addressed for session calculi, both binary [33, 23] and multiparty [14, 34, 28, 24] (see Section 6 for more discussion on related work).

When reversing a structured interaction, one has to face the problem of preserving consistency of the global state: if one of the partners triggers a rollback, then all its communicating partners should roll back accordingly. This is where session types come to the rescue, with their precise specification of the functionality of communications (sender, receiver and message), and of the order in which they should occur.

We present a calculus for concurrent reversible multiparty sessions, which extends previous proposals in several ways. First of all, our protocols are more general than those specified by standard multiparty session types [18]: concurrent communications are allowed both between disjoint groups of participants, and, in a controlled way that excludes auto-concurrency, also within participants themselves. More specifically, we relax the linearity constraint of standard multiparty session types, to allow protocol participants to perform communications in parallel, as long as they do not generate races. To enable the control flows to join again after a bunch of parallel interactions, besides parallel composition we also introduce full sequential composition in the syntax of both types and processes.

Moreover, our calculus gives a compact representation of the past of processes and types, which facilitates the definition of rollback, and it implements a fine-tuned strategy for backward computation, which is geared towards achieving compliance.

The main contributions of our paper may be summarised as follows:

- the introduction of concurrent session types in multiparty session frameworks;
- the proposal of a reversible session calculus which is more general than the existing ones, and which conservatively extends them, preserving the expected properties of session fidelity, forward and backward progress, and causal consistency;
- a fine-tuned strategy for rollback to checkpointed choices, which can only be triggered by choice leaders in predefined states of the computation, leading back to the choice state stripped off the unsuccessful path, so as redirect the computation towards alternative, potentially successful, paths.

The rest of the paper is organised as follows. In Section 2 we present our running example. In Section 3 we define the syntax and operational semantics – both forward and backward – of our calculus. In Section 4 we introduce our extended syntax for global types and session types and we establish well-formedness conditions for global types. Section 5 presents our type system and proves its soundness, namely that it ensures the expected semantic properties. We conclude in Section 6 with some discussion on related and future work.

## 2 Travel protocol example

To illustrate our approach, we present a simple travel protocol, involving three parties, a Researcher $R$, a Student $S$ and an Airline company $A$. This protocol, henceforth called the RSA protocol, will serve as our running example throughout the paper.
When graphically representing protocols, we shall use two kinds of graphs: the first one, called *global graph*, is borrowed from [13] and gives a global description of the protocol; the second one, which we call *message sequence graph*, is closer to message sequence charts and emphasises the control flows of the individual partners. In both cases, diamonds will be used as delimiters for the branching construct, and squares as delimiters for the parallel construct.

In the global graph, boxes represent communications and their whole functionality: sender, receiver and message. In the message sequence graph, the control flow of the protocol is split into the individual contributions of the partners: each partner is denoted by a vertical line; communications are represented by arrows from the sender to the receiver, labelled by messages; and branching and parallel constructs are replicated for each involved partner.

The RSA protocol is depicted in Figure 1a and Figure 1b according to these conventions. Let us now informally describe the protocol. Suppose that R and S wish to travel together to some conference, and that R has date constraints but is flexible about prices, while S has funding constraints but is flexible about dates. The protocol starts by R sending to S her chosen date. Then S sends two parallel flight requests to A: one for the date chosen by R, and one for more flexible dates. The airline A replies by sending in parallel the respective fares: a higher fare for the fixed date, and a lower fare for the flexible date. At this point, if S can afford the higher fare, she will book two tickets at this fare, then send a message done+ to R, and the whole interaction will end here.

Suppose now that S cannot afford the higher fare. In this case, she will propose to R the alternative date corresponding to the cheaper fare. If R cannot travel at this date she replies no. Then S will book a single ticket for herself at the cheaper fare, and the interaction terminates. If instead this date is suitable for R, she replies yes to S. Now S books two tickets at the cheaper fare and sends a message done- to R. Then R informs S that the hotel...
booking has been changed to fit the new date (supposing R had previously booked the hotel for her chosen date), thereby closing the interaction.

So far we have described only the forward behaviour of the session. Suppose now that after proposing the alternative date to R and receiving R’s agreement, S discovers that this date is not suitable anymore. Then S will trigger a rollback to her internal choice with checkpoint label C, and R will have to roll back to her corresponding external choice with checkpoint label C. At this point, since the first branch of the choice has been already explored with no success, S will have to choose the second branch and book the more expensive flight. On her side, R could decide to roll back to her internal choice with label C’ if she discovers that the hotel is fully booked at the alternative date, and then both S and A will have to roll back to their corresponding external choices with label C’.

In our calculus, only “the leader” of a choice, i.e., the participant who solved the choice by sending the first message, will be authorised to trigger a rollback. In the RSA protocol, the leader of the first choice is S, while the leader of the second choice is R. In the message sequence graph, the leader is distinguished by the fact that her choice has an overlined checkpoint. In the global graph, which collapses the distributed structure of the interaction, the choice leader is the sender in the first communication after the choice. In fact, the graph of Figure 1b can always be derived from that of Figure 1a through a projection operation.

3 Calculus

In this section, we introduce the syntax and semantics of our calculus. As usual in session calculi, we distinguish between user processes, which have not started to be executed yet, and runtime processes. The executed part of runtime processes will be marked with hats: this extension is instrumental to reversing computations. Processes may be decorated by checkpoint labels, marking them as possible rollback points. Our syntax generalises that of standard multiparty session calculi [18, 12], featuring the additional operators of parallel and sequential composition (the latter replacing the prefixing operator).

We assume the following base sets: messages, ranged over by λ, λ’, . . . and forming the set Msg; checkpoint labels, ranged over by C, C’ and forming the set ChLa; and session participants, ranged over by p, q, r and forming the set Part.

We use δ to range over general sets of checkpoint labels, and Δ to stand for either δ or δ extended by exactly one overlined checkpoint label:

\[ \delta ::= \emptyset \mid \delta, C \]
\[ \Delta ::= \delta \mid \delta, C' \]

Sets of checkpoint labels are associated with choices. More precisely, an overlined checkpoint label can only be associated with an internal choice and is said to be active. A simple checkpoint label is passive and may be associated with both internal and external choices. Intuitively, an overlined checkpoint label is the handle of a backward move: a participant who crossed an internal choice (henceforth also called the choice leader) with an active checkpoint label, and then proceeded in the computation, may decide to return to that choice whenever she has the ability to send a message. Within a network, this backward move of the choice leader will have to be matched by backward moves of all the participants who did some action after that checkpoint label.

Let \( \pi \in \{p?\lambda, p!\lambda \mid p \in \text{Part}, \lambda \in \text{Msg} \} \) denote an atomic action, namely a directed input or output action. An atomic action can bear a hat, in which case it represents an already executed or past action. We use \( \bar{\pi} \) to stand for either \( \pi \) or \( \bar{\pi} \). External and internal choices can also bear hats, indicating that one branch has been chosen. We use \( \sum \) to stand for either \( \sum \) or \( \sum \), and similarly for \( \bigoplus \).
Definition 1. Processes are defined by:

\[
P ::= \sum_{i \in I} \pi_i P_i + \bigoplus_{i \in I} \pi_i P_i + \Delta P + P|P + P; P + \mu X.P + X | \text{skip}
\]

Processes without and with hats are called respectively user processes and runtime processes.

We will omit empty sets of checkpoint labels, choice symbols in one-branch choices, and trailing skip processes. External and internal choices are assumed to be associative, commutative, idempotent, and non-empty (except when combined with binary choices in evaluation contexts or finished processes, see below). A choice with a single branch may be either an input process or an output process. Parallel composition is associative and commutative, with neutral element \text{skip}. Sequential composition is associative, with neutral element \text{skip}. The operators have the following precedence: ‘;’, ‘+’, ‘⊕’, ‘|’.

Following [27], we require recursion to be:

- guarded, i.e. a recursion variable \( X \) can only appear free in the second argument of a sequential composition whose first argument is different from \text{skip};
- sequential, i.e. a recursion variable \( X \) cannot occur free in any branch of a parallel composition.

Processes are treated equi-recursively, i.e. they are identified with their generated tree [31].

The typing rules of Section 5 will ensure the following well-formedness conditions for processes:

1. External choices are between input processes;
2. Internal choices are between output processes;
3. A choice without a hat has branches without hats;
4. A choice with a hat has exactly one branch with hats;
5. Only choices are decorated with sets of checkpoint labels;
6. Only internal choices are decorated with an active checkpoint label.

Conditions 1 and 2 reflect the active role of outputs and the passive role of inputs. Conditions 3 and 4 express the fact that executing a choice amounts to executing one of its branches. Conditions 5 and 6 specify that choices are the only return points for backward reductions. In the following we will take advantage of these restrictions to simplify definitions.

In a full-fledged calculus, messages would carry values, namely they would be of the form \( \lambda(i.P) \) or \( \mu X.P \). Here, for simplicity we consider only pure messages.

Networks are parallel compositions of pairs \( p[P] \), where participant \( p \) has behaviour \( P \).

Definition 2. Networks are defined by:

\[
N ::= p[P] | N || N
\]

The operator \( || \) is associative and commutative, with neutral element \( p[\text{skip}] \) for each \( p \).

The operational semantics is given by two LTSs, one for processes and one for networks. In the LTS for processes, forward transitions have the form \( P \xrightarrow{\pi} P' \) and backward transitions have the form \( P \xleftarrow{\pi} P' \) or \( P \xleftarrow{\lambda} P' \). We define \( P \xrightarrow{\pi} P' \) if \( P \xleftarrow{\pi} P' \) for some \( \pi \).

In the LTS for networks, forward and backward transitions have respectively the form \( N \xrightarrow{p,\lambda} N' \) and \( N \xleftarrow{C} N' \).

Definition 3. Evaluation contexts \( E \) are defined by:

\[
E ::= \delta(\sum_{i \in I} \pi_i P_i; \bigoplus_{i \in I} \pi_i P_i; E) + \Delta(\bigoplus_{i \in I} \pi_i P_i; \bigoplus_{i \in I} \pi_i F_i; E) + E|P + E;P + E;F + F;E + []
\]

The definition of evaluation context ensures that in a sequential composition the evaluation of the second component can only start when the first is a finished process \( F \), defined by:

\[
F ::= \delta(\sum_{i \in I} \pi_i P_i; \bigoplus_{i \in I} \pi_i F_i) + \Delta(\bigoplus_{i \in I} \pi_i P_i; \bigoplus_{i \in I} \pi_i F_i) + F|F + F;F + \text{skip}
\]
30:6 Concurrent Reversible Sessions

The LTSs for processes and networks are given in Figure 2. Rules [ExtCh] and [IntCh] allow an action to be extracted from one of the summands, as usual, but instead of discarding the other summands they record the fact that the choice has been crossed by marking the choice operator with a hat. With this technique, inspired by [6] and already used for reversible computations in [24], all the dynamic operators are turned into static operators, and nothing is lost of the original user process. Notice that when \( I = \{ j \} \) these rules become \( \pi_j; P_j \xrightarrow{\delta} \pi_j ; P_j \). Rule [BackCKT] is the main backward rule: it applies to a past internal choice, where one branch has been partially executed, and it allows the process to roll back to the original choice where the executed branch is removed. For this to be possible, the choice must have at least one non executed branch \( P_i \) and a set of checkpoint labels containing an overlaid label \( \mathcal{C} \), which will label the back transition. This is essential to ensure (by means of typing) that the choice leader will be the only one who can decide to roll back to this choice. The condition \( P \downarrow_{\text{out}} \) means that in order to trigger a rollback, \( P \) should be “in lead” again, namely able to do an output.

Rule [BackP], where \( S \) denotes either \( \sum_{i \in I} P_i \), or \( \bigoplus_{i \in I} P_i \), is needed to allow the remaining participants to roll back. The mapping \( \downarrow \) erases hats from processes, yielding user processes, i.e. \( [\pi; P] = \pi; P \) and \( i \) acts homomorphically otherwise. Note that the rollback rules can only be applied to processes that are not user processes: in particular, one branch can be erased only if at least one of its actions has been executed. An evaluation context \( \mathcal{E} \) is \( \text{ok} \) for \( \mathcal{C} \) (\( C \)) if \( C \notin \Delta \) (\( C \in \Delta \)) whenever \( \mathcal{E} \) is a sub-context of a context of the shape \( \Delta \sum_{i \in I} P_i \uplus \widehat{\pi}; \mathcal{E}' \) or \( \Delta \bigoplus_{i \in I} P_i \uplus \widehat{\pi}; \mathcal{E}' \). We use this condition in rules [CtBA] and [CtBP] to assure that all participants involved in a recursion go back to the same checkpoint, namely to the first one, as in [28]. This is needed to assure subject reduction, see Example 8.

Rule [Com] is standard. We write \( P \not\triangleright j\) if neither Rule [BackP] nor Rule [CtBP] (with label \( C \)) can be applied to \( P \). This means that \( C \) can only occur in user processes within \( P \).
In a well-typed network, Rule [BACK] will make participant p roll back to an internal choice and moreover, all participants that can roll back to this choice will do so in the same step. This will be the basis for our soundness result in Section 5. A direct implementation of this rule is clearly unrealistic. To that purpose, asynchronous communications including rollback messages should be used, as in [24]. As expected, sequences of network transitions generate traces, which are words over the infinite alphabet \{pλq \mid p, q \in \text{Part}, λ \in \text{Msg}\} \cup \text{ChLa}.

When the labels of transitions are not relevant, we write them simply as \(\rightarrow\) and \(\rightarrow^*\). In this case, we use \(\rightarrow^*\) to denote the reflexive and transitive closure of \(\rightarrow\) and \(\rightarrow^*\) to denote the reflexive and transitive closure of \(\rightarrow\) and \(\rightarrow^*\).

Note that our semantics does not enforce any scheduling policy. Hence, a network may generate an infinite trace that does not involve all participants. For instance, the network:

\[ p[\mu X_1.q[\lambda_1:X_1 | \mu X_2.r[\lambda_2:X_2] | q[\mu X_3.p[?\lambda_1:X_3] | r[\mu X_4.p[?\lambda_2:X_4]] \]

may generate the infinite trace \(pλq.pλq\ldots\) in a more elaborate calculus, we could impose a fair scheduling policy, forcing p to communicate alternately with q and r, and in general, no communication to be iterated until each participant has communicated at least once.

A network for our RSA travel protocol in Section 2 is \(N_{in} = R[\{P^R\} \parallel S\{P^S\} \parallel A\{P^A\}]\), defined as follows (where messages are abbreviated in a programming language style):

- The process \(P^S\) is \(R!\text{dt}:(P^S_1 \parallel P^S_2); (P^S_3 \parallel P^S_4)\) where
  \[ P^S_1 = A!\text{fx} \text{dt}:A!\text{fr} P^S_2 = A!\text{fd} \text{dt}:A!\text{fr} \text{Mn} P^S_3 = A!\text{al} \text{dt}:(P^S_{3'}) + P^S_0 P^S_4 = A!\text{al} \text{dt}:A!\text{fr} P^S_6 = A!\text{al} \text{dt}:A!\text{fr} P^S_0 = A!\text{al} \text{dt}:A!\text{fr} P^S_6 = A!\text{al} \text{dt}:A!\text{fr} \text{Mn} P^S_1 = A!\text{al} \text{dt}:A!\text{fr} P^S_6 = A!\text{al} \text{dt}:A!\text{fr} \text{Mn} \]

- The process \(P^R\) is \(S!\text{dt}:(P^R_1 \parallel S?\text{dn} P^1)\) where
  \[ P^R_1 = S?\text{al} \text{dt}:(P^R_{1'}) \parallel S?\text{no} P^R_2 = S?\text{al} \text{dt}:(P^R_{2'}) \parallel S?\text{no} \]

- The process \(P^A\) is \((P^A_1 \parallel P^A_2); (C,C') \sum_{i \in \{3,4,5\}} P^A_i\) where
  \[ P^A_1 = S?\text{al} \text{dt}:(P^A_{1'}) \parallel S?\text{dn} P^1 \]
  \[ P^A_2 = S?\text{al} \text{dt}:(P^A_{2'}) \parallel S?\text{dn} P^2 \]
  \[ P^A_3 = S?\text{al} \text{dt}:(P^A_{3'}) \parallel S?\text{dn} P^3 \]
  \[ P^A_4 = S?\text{al} \text{dt}:(P^A_{4'}) \parallel S?\text{dn} P^4 \]

In the computation below, we denote with \(\hat{P}\) the process \(P\) entirely marked with hats.

\[
\begin{align*}
N_{in} & \quad \xrightarrow{R!\text{dt} S} \quad R[S!\text{dt}:(P^R_1 \parallel S?\text{dn} P^1)] \parallel S[R\!\text{dt}:(P^S_1 \parallel P^S_2); (P^S_{3'}) \parallel P^S_4)] \parallel A\{P^A\} \\
& \quad \xrightarrow{\rightarrow^*} \quad R[S!\text{dt}:(P^R_1 \parallel S?\text{dn} P^1)] \parallel S[\hat{F}^S]: (P^S_{3'}) \parallel P^S_4)] \parallel A\{\hat{Q}^A\}
\end{align*}
\]

\[
\begin{align*}
& \quad \xrightarrow{S?\text{al} \text{dt} R} \quad R[S!\text{dt}:(S?\text{al} \text{dt}:(P^R_{1'} \parallel S?\text{dn} P^1)) \parallel S[\hat{F}^S]: (P^S_{3'}) \parallel P^S_4)] \parallel A\{\hat{Q}^A\} \\
& \quad \xrightarrow{\rightarrow^*} \quad R[S!\text{dt}:(S?\text{al} \text{dt}:(S?\text{al} \text{dt}:(P^R_{1'} \parallel S?\text{dn} P^1) \parallel S[\hat{F}^S]: (P^S_{3'}) \parallel P^S_4)] \parallel A\{\hat{Q}^A\}
\]
\]

Note that the last network differs from that in Line 2 only by the absence of process \(P^S_3\).

4 Global Types and Session Types

A multiparty session is a series of communications among a fixed number of participants [18], which follows a predefined protocol specified by a global type.
We use $\gamma$ to denote either the empty set or a singleton made of a checkpoint label:

$$\gamma ::= \emptyset \mid \{C\}$$

Sets $\gamma$ will be associated with choices in global types.

Let $\alpha^p \in \{ p \overset{\lambda}{\to} q \mid q \in \text{Part}, \lambda \in \text{Msg}\}$ denote an atomic communication with sender $p$. An atomic communication can bear a hat, in notation $\hat{\alpha}$, in which case it represents an executed or past communication. The symbol $\hat{\alpha}$ stands for either $\alpha^p$ or $\hat{\alpha}^p$.

Global types $G$ are built from choices among atomic communications with the same sender, with the constructs of parallel composition, sequential composition, and recursion. A global type $K$ specifies an interaction that is still to start. On the opposite, a global type $H$ specifies a completely executed interaction, in which there is one path entirely marked with hats. A general global type $G$ specifies a partially executed interaction, whose executed part (history) is specified by subterms $H$, while the parts that have been discarded in choices or remain to be executed are specified by subterms $K$.

**Definition 4.** **Global types** $G$ are defined by:

$$K ::= \gamma \oplus_{i \in I} \alpha^p_i; K_i \mid K \parallel K \mid K; K \mid \mu t. K \mid t \mid \text{Skip}$$

$$H ::= \gamma \bigoplus_{i \in I} \alpha^p_i; K_i \parallel \hat{\alpha}^p \parallel H \parallel H \parallel H; H \mid \text{Skip}$$

$$G ::= K \mid \gamma \bigoplus_{i \in I} \alpha^p_i; K_i \parallel \hat{\alpha}^p \parallel H \parallel G \parallel G \mid G; K \mid K; G$$

The choice operator $\oplus$ is $n$-ary, commutative and idempotent. We use $\bigoplus$ for either $\oplus$ or $\hat{\oplus}$. In the binary case we write $\alpha^p_1; G \bigoplus \alpha^p_2; G'$. The notation $\gamma \bigoplus_{i \in I} \alpha^p_i; K_i \bigoplus \hat{\alpha}^p \parallel H \parallel G$ stands for a choice where the branch initiating with $\hat{\alpha}^p$ has started to be executed. By abuse of notation, we will write $\gamma \bigoplus_{i \in I} \alpha^p_i; G_i$ for either $\bigoplus_{i \in I} \alpha^p_i; G_i$ or $\bigoplus_{i \in I \setminus \{j\}} \alpha^p_i; G_i \bigoplus \hat{\alpha}^p_j \parallel G_j$. When the checkpoint set $\gamma$ associated with a choice is empty, we omit it. We call rooted interaction a subterm $\hat{\alpha}^p; G$. Hence a type $\gamma \bigoplus_{i \in I} \alpha^p_i; G_i$ is a choice among a number of rooted interactions with the same sender, at most one of which bears a hat. If $I$ is a singleton we write $\alpha^p; G$. So $\hat{\alpha}^p; G$ can denote either a rooted interaction or a choice with a single branch (the context will disambiguate if needed). We use $\text{pa}(G)$ to denote the set of participants of $G$, namely all $p, q$ such that $p \overset{\lambda}{\to} q$ occurs in $G$.

As for processes, parallel and sequential composition are associative, with neutral element $\text{Skip}$, parallel composition is commutative, and recursion is guarded, sequential, and treated equi-recursively. The operators have the following precedence: ‘;’, ‘$\parallel$’, ‘$\oplus$’.

**Session types** are projections of global types onto participants. They represent the contributions of individual participants to the session. The projection of a choice yields a union for the choice leader and intersections for the receivers. Checkpoint labels of global types are preserved by the projection onto session types, and the checkpoint label of the choice leader is distinguished by overlining it.

We now define **session pre-types**, which are a superset of session types. In this paper, session types will only be defined as projections of well-formed global types, see Definition 7. (There is no circularity since global type well-formedness does not depend on session types.) Session pre-types are obtained from processes by replacing external and internal choices with intersections and unions, $X$ with $t$ and $\text{skip}$ with $\text{Skip}$, with similar conventions.

**Definition 5.** **Session pre-types** are defined by:

$$T ::= \bigwedge_{i \in I} \hat{\alpha}^p_i; T_i \mid \bigvee_{i \in I} \hat{\alpha}^p_i; T_i \mid \Delta T \mid T \parallel T \mid T; T \mid \mu t. T \mid t \mid \text{Skip}$$
We call an intersection $\bigwedge_{i \in I} p_i ? \lambda_i ; T_i$ non-ambiguous if its initial inputs are all distinct, namely if $i \neq j$ implies either $p_i \neq p_j$ or $\lambda_i \neq \lambda_j$. Projectability will require non-ambiguity of intersections. Our definition of projection (see Figure 3) is more liberal than in other session calculi, since we have an extended syntax for global types and we want to maximise the set of global types that are projectable. In particular, we allow equal messages between the same pair of participants in choices (this is usually forbidden [12, 28]). In any case, we want projection to ensure that in a global choice, the choice leader makes the decision and all the other participants act accordingly. We do so by requiring that, for any participant except the choice leader, the set of projections of the choice branches on that participant, say $\{ T_i \mid i \in I \}$, be consistent, i.e., the join of the types in the set, $\bigsqcup_{i \in I} T_i$, be defined.

The join $\bigsqcup_{i \in I} T_i$ is a partial operator, which checks that the $T_i$’s are compatible behaviours and then combines them into a single session type. (In the definition of join we may assume that all the $T_i$’s are projections of global types.) Intuitively, $\bigsqcup_{i \in I} T_i$ is defined if the concerned participant either has the same behaviour in all $T_i$, or, if this is not the case, if she receives a message that “notifies” her about the chosen $T_i$ before she starts differentiating her behaviour.

If one of the $T_i$’s is an intersection of input behaviours, then so must be all the other $T_i$’s.

To join intersection types, we define an auxiliary operator $\{\}$, which takes a non-ambiguous intersection $\bigwedge_{i \in I} p_i ? \lambda_i ; T_i$ and a session type $p ? \lambda ; T$, and combines them, if possible:

$$\bigwedge_{i \in I} p_i ? \lambda_i ; T_i \bigcup p ? \lambda ; T = \begin{cases} \bigwedge_{i \in I} p_i ? \lambda_i ; T_i \land p ? \lambda ; T & \text{if } p_i \neq p \text{ or } \lambda_i \neq \lambda \text{ for all } i \in I \\ \bigwedge_{i \in I \setminus \{j\}} p_i ? \lambda_i ; T_i \land (p_j ? \lambda_j \cup p ? \lambda) ; T \bigcup T_j & \text{if } p_i = p \\
\text{and } \lambda_j = \lambda (j \in I) \end{cases}$$

where $p_j ? \lambda \cup p ? \lambda = p ? \lambda \cup p ? \lambda = p ? \lambda$ and $p ? \lambda \cup p ? \lambda = p ? \lambda$ and the obtained intersections have hats if one of the input behaviours has a hat. If the prefix $p ? \lambda$ is different from all the $p_i ? \lambda_i$, then $\{\}$ produces the intersection of the two types. In this case, receiving message $\lambda$ from $p$ amounts to being notified of the choice of branch $T$. If instead $p ? \lambda = p_j ? \lambda_j$ for some $j \in I$, then we try to combine the types $p_j ? \lambda_j ; T_j$ and $p ? \lambda ; T$ by factoring out their common prefix and producing $p ? \lambda ; T \bigcup T_j$ (where $p ? \lambda$ has a hat if one of the two building prefixes has a hat), so that the resulting intersection is again non-ambiguous. If $T \bigcup T_j$ is defined, then $T$ and $T_j$ will be discriminated later if they are different, see $G_1$ in Example 6.

To define $\{\}$ on two intersection types, we just iterate the above definition on the members of the intersections. In a similar way, we can extend $\{\}$ to a set of intersection types.

We may now formally define the join of a set of session types.

$$\bigsqcup_{i \in I} T_i = \begin{cases} \Delta \{\} \bigsqcup_{i \neq j \in I, j \in I} p_{i,j} ? \lambda_{i,j} ; T_{i,j} & \text{if } T_i = \Delta \bigwedge_{j \in I} p_{i,j} ? \lambda_{i,j} ; T_{i,j} \text{ for all } i \in I \\
\text{and } \Delta = \bigcup_{i \in I} \Delta_i \\
T_i \bigsqcup_{i \in I} T_i' & \text{if } T_i = T_i' \text{ for all } i \in I \\
\text{Skip} & \text{if } T_i = \text{Skip} \text{ for all } i \in I \end{cases}$$

The definitions of $\bigsqcup_{i \in I} T_i$ and $\{\}$ are mutually recursive. Let us examine the three clauses. If all $T_i$’s are intersections of session types, then we combine them with $\{\}$ as explained above. The checkpoint set of the resulting intersection is the union of the checkpoint sets of the $T_i$’s.

If all $T_i$’s start with the same interaction $T$, and their continuations $T_i'$’s are consistent, then we factor out $T$ and we produce the session type $T$ followed by the join of the $T_i'$’s.

If some $T_i$ is Skip, it means that the participant terminates in the branch $T_i$. Then it must terminate in all branches, so all $T_i$’s must be Skip.

The join is not defined between intersections and unions, nor between unions with different prefixes. Our join extends that of [12], to deal with parallel and sequential composition.
To define projection we need one last auxiliary operator, the prefixing of a session pre-type \( T \) by a set \( \gamma \) of at most one checkpoint label (notation \( \gamma[T] \)), defined by:

\[
\gamma[T]_0|T_1 = \gamma[T_0]|\gamma[T_1] \quad \gamma[\Delta T] = \gamma[\Delta] T \quad \text{if} \quad \gamma \cap \Delta = \emptyset
\]

Prefixing adds \( \gamma \) to the first (possibly empty) set \( \Delta \) found in \( T \). Intuitively, the checkpoint labels that are spread along successive choices in the global type may get grouped together on a single local choice when projected on participants. We do not need to define \( \gamma[t] \) since recursion is guarded. The condition in the clause for \( \gamma[\Delta T] \) is needed to avoid checkpoint labels which can never be used. This condition rules out for instance the global type:

\[
\lambda \gamma \{ C \ | \ p \overset{\lambda_1}{\rightarrow} q; p ; p \overset{\lambda_1}{\rightarrow} r \}
\]

in which no backward reduction could return to the innermost occurrence of \( C \). This is due to the condition on evaluation contexts in rules [CtBA] and [CtBP].

The projection of global types uses the projections of rooted interactions, see the first line of Figure 3. The projection of a choice is a union for the sending participant, and it is otherwise computed as the join of the projections on the branches. With \( \overline{\gamma} \) we denote \( \{\overline{\gamma}\} \) if \( \gamma = \{\overline{\gamma}\} \) and \( \emptyset \) if \( \gamma = \emptyset \). The join operation can be undefined, and therefore also the projection of global types can be undefined. Since \( \text{Skip} \models \emptyset \) T is defined only if \( T = \text{Skip} \), the definition of projection ensures that all the branches of a choice \( \gamma \overline{\bigcup} i \in I \alpha_i^2; G_i \) have the same participants, i.e., \( p a(\alpha_i^2; G_i) = p a(\alpha_j^2; G_j) \) for all \( i, j \in I \). The omitted cases in Figure 3 are either standard (recursion and Skip) or homomorphic projections (parallel and sequential composition). Notice that projection respects hats and checkpoint labels.

\[\text{Example 6.}\] Let \( G_1 = ( p \overset{\lambda_1}{\rightarrow} q; p \overset{\lambda_1}{\rightarrow} q ; q \overset{\lambda_2}{\rightarrow} r ) \| \overset{\lambda_3}{\rightarrow} ( p \overset{\lambda_3}{\rightarrow} q; p \overset{\lambda_3}{\rightarrow} q; q \overset{\lambda_4}{\rightarrow} r ). \]

The projections of \( G_1 \) onto its participants are: \( G_1 | p = ( q \overset{\lambda_3}{\rightarrow} q ; q \overset{\lambda_3}{\rightarrow} q ) \, \overset{\lambda_3}{\rightarrow} ( q \overset{\lambda_3}{\rightarrow} q; q \overset{\lambda_3}{\rightarrow} q ) \), \( G_1 | q = p \overset{\lambda_2}{\rightarrow} q; \overset{\lambda_2}{\rightarrow} q \overset{\lambda_2}{\rightarrow} q; \overset{\lambda_2}{\rightarrow} q \overset{\lambda_2}{\rightarrow} q \) and \( G_1 | r = q \overset{\lambda_2}{\rightarrow} q \overset{\lambda_2}{\rightarrow} q \overset{\lambda_2}{\rightarrow} q \overset{\lambda_2}{\rightarrow} q \). Even though \( q \) receives the same message \( \lambda_3 \) from \( p \) in the two branches of the choice, before sending two different messages to \( r \) it receives a second message from \( p \), which identifies the chosen branch. Also the global type \( G_2 = ( p \overset{\lambda_1}{\rightarrow} q; r \overset{\lambda_3}{\rightarrow} q) \| \overset{\lambda_3}{\rightarrow} ( p \overset{\lambda_3}{\rightarrow} q; r \overset{\lambda_3}{\rightarrow} q ) \) is projectable, since \( r \) sends the same message \( \lambda_3 \) in both branches.

Instead, the global type \( G_3 = ( p \overset{\lambda_1}{\rightarrow} q; q \overset{\lambda_2}{\rightarrow} r ) \| \overset{\lambda_3}{\rightarrow} ( p \overset{\lambda_3}{\rightarrow} q; q \overset{\lambda_2}{\rightarrow} r ) \) is not projectable on the participant \( q \) in fact \( ( p \overset{\lambda_2}{\rightarrow} ; r \overset{\lambda_3}{\rightarrow} ) \| \overset{\lambda_1}{\rightarrow} ( p \overset{\lambda_2}{\rightarrow} ; r \overset{\lambda_1}{\rightarrow} ) \) is not defined, since it yields \( p \overset{\lambda_2}{\rightarrow} ; ( r \overset{\lambda_1}{\rightarrow} ) \| \overset{\lambda_1}{\rightarrow} ( r \overset{\lambda_2}{\rightarrow} ) \) and \( r \overset{\lambda_1}{\rightarrow} \| r \overset{\lambda_2}{\rightarrow} \) is not defined. This is justified since \( q \) should send two different messages to \( r \) in the two branches, while no previous communication identifies the chosen branch.

The next example features a checkpoint label set containing both a simple label and an overlined one. Let \( G_4 = p \overset{\lambda_1}{\rightarrow} q ; p \overset{\lambda_1}{\rightarrow} r ; G_5 \overset{\{C\}}{\|} p \overset{\lambda_3}{\rightarrow} q ; p \overset{\lambda_3}{\rightarrow} r ; G_5 \) where \( G_5 = \)
On the other hand, the projections of branches do not have equal senders, equal messages and equal receivers. Indeed, if we allowed through some communications. For parallel global types, instead, we require that different that the behaviour of a participant is either independent of the chosen branch or identifiable have equal senders, equal messages and equal receivers. Our definition of projection ensures type would allow her to roll back to the choice labelled by first interaction. Then participant λ message Here there are two parallel interactions, both involving participants λ, q and r. Assume that message λ₁ has been exchanged in the first interaction, and message λ₃ has been exchanged in the second. Now participant p may want to roll back to the choice labelled by {C} in the first interaction. Then participant q should roll back to the same choice, while the above type would allow her to roll back to the choice labelled by {C} in the second interaction.

As regards messages, we already observed that we allow different branches of a choice to have equal senders, equal messages and equal receivers. Our definition of projection ensures that the behaviour of a participant is either independent of the chosen branch or identifiable through some communications. For parallel global types, instead, we require that different branches do not have equal senders, equal messages and equal receivers. Indeed, if we allowed
two parallel communications with equal messages between the same pair of participants, then we could have a global type \( p \xrightarrow{\lambda} q \colon G_1 \parallel p \xrightarrow{\lambda} q \colon G_2 \), and processes implementing participants \( p \) and \( q \) could “cross” their communications.

A global type meeting the above two requirements on messages and checkpoint labels is called respectively message well formed and checkpoint label well formed.

Last but not least, following \([7, 26]\) we require that the order of communications prescribed by a global type be witnessed by at least one of the session participants. This excludes, for instance, the global type \( p \xrightarrow{\lambda} q \parallel r \xrightarrow{\lambda} s \), since the two communications have disjoint sets of participants, and therefore they should be independent. The correct global type representing this situation is \( p \xrightarrow{\lambda} q \parallel r \xrightarrow{\lambda} s \). This condition is necessary for a global type to be implementable by a collection of processes, and we refer to \([7]\) for further justifications of this choice. It is easy to formalise this requirement as a well-formedness condition on the set of traces generated by global types, defined in the standard way.

To sum up, we define well-formedness of global types as follows.

**Definition 7** (Well-formed global types). A global type is well formed when:

1. Its projections are defined on all participants;
2. It is sequentially well formed, message well formed and checkpoint label well formed;
3. Its set of traces is well formed.

In the following we will consider only well-formed global types.

## 5 Type System and Soundness

The shape of typing judgements is \( \Gamma \vdash P : T \), where the environment \( \Gamma \) associates process variables with session types: \( \Gamma ::= \emptyset | \Gamma, X : T \). Process typing exploits the correspondence between external choices and intersections, internal choices and unions. Figure 4 shows the interesting rules. Typing respects hats, in rules \([t-ExtCh], [t-IntCh], [t-ExtCh-RT], [t-IntCh-RT]\) intersections and unions have hats if choices have hats. Subtyping \( \subseteq \) on session types takes into account the standard rules for intersection and union and preserves hats, while checkpoint sets can decrease with deletion of types in intersections and increase with addition of types in unions. Subtyping is used in a standard subsumption rule (omitted). The other omitted rules for processes are just homomorphisms as expected from the syntax of processes and session types. Rule \([t-Net]\) is the only rule for typing networks: it
This example shows also why it would not be possible to roll back to the last checkpoints in
given at the end of Section 3.

Notice that Theorems 9 and 10 imply Subject Reduction for
of well-typed nets and the evolution of their types along forward and backward computations.

The relation between the type systems \( \vdash \) and \( \vdash * \) is expressed by Theorem 9, whose
proof uses standard Inversion Lemmas. The property of Subject Reduction for \( \vdash * \) is then
established by Theorem 10. Its proof relies on the close correspondence between the evolution
of well-typed nets and the evolution of their types along forward and backward computations.

\textbf{Theorem 9.} If \( \vdash * \mathbf{N} : \mathbf{G} \), then \( \vdash \mathbf{N} : \mathbf{G} \). If \( \vdash \mathbf{N} : \mathbf{G} \), then \( \vdash * \mathbf{N} : \mathbf{G}' \) for some \( \mathbf{G}' \).

\textbf{Theorem 10 (Subject Reduction).} \( \vdash * \mathbf{N} : \mathbf{G} \) and \( \mathbf{N} \overset{\lambda \rightarrow}{\rightarrow} \mathbf{N}' \) imply \( \vdash * \mathbf{N}' : \mathbf{G}' \) for some \( \mathbf{G}' \).

Notice that Theorems 9 and 10 imply Subject Reduction for \( \vdash \).

A standard property enforced by session types is Session Fidelity: all communications
occur as specified by global types. To deal with backward reductions we introduce a mapping
\( \vdash \) on global types, which erases all communications with hats and all discarded branches
of choices. A checkpoint label \( C \) is \textit{alive} in \( \mathbf{G} \) if \( \vdash \mathbf{G} \) contains a choice \( \vdash (C) \vdash \mathbf{p} \overset{\lambda \rightarrow}{\rightarrow} \mathbf{q}; \mathbf{G}_{i} \),
with more than one branch, such that for some \( i \in I, \lambda, \mathbf{q} \), the global type \( \mathbf{G}_{i} \) contains \( \mathbf{p} \overset{\lambda \rightarrow}{\rightarrow} \mathbf{q} \).
Theorem 11 (Session Fidelity). If \( \vdash N : G \), then all traces in forward computations of \( N \) are suffixes of traces of \( G \). Moreover if \( N \rightarrow^* N' \bowtie N'' \), then \( C \) is alive in \( G \).

We discuss now forward and backward progress.

Theorem 12 (Forward Progress). If \( \vdash P[E[P]] \parallel N : G \) with \( P \) user process, \( P \neq \text{skip} \), then there are \( P', N' \) such that \( P[E[P']] \parallel N \rightarrow^* P[E[P']] \parallel N' \) and \( P' \) has one hat.

Notice that the standard formulation of progress [12], which requires that each input/output that is persistently offered be eventually consumed, is an easy consequence of this theorem.

Theorem 13 (Backward Progress). If \( \vdash^* N : G \) and \( C \) is alive in \( G \), then there is \( N' \) such that \( N \rightarrow^* N' \) and \( N' \bowtie N'' \).

We end this section with a remark on causal consistency and full reversibility. Clearly, our calculus enjoys causal consistency since a rollback to a checkpointed choice cancels all communications done after that checkpoint. Instead, full reversibility does not hold, since rollbacks erase explored branches in internal choices.

6 Related Work and Conclusion

Since the seminal work by Danos and Krivine on reversible CCS [9], reversible computation has been widely studied in process calculi. In [29], Phillips and Ulidowski proposed a method for reversing process operators defined in a general SOS format, and noted that thread identifiers and histories were needed to record the past of computations. In [21], Lanese et al. defined a reversible variant of the higher-order \( \pi \)-calculus, using tags to identify threads, and explicit memory processes. This calculus was enriched with a fine-grained rollback primitive in [20]. In [8], Cristescu et al. proposed a causal semantic model for the reversible \( \pi \)-calculus.

Reversibility for structured communications was first studied in [10, 11, 19], where transactions with rollback and coordinated checkpoints were modelled in an extended CCS. More recently, reversibility has been incorporated into contracts [1, 4] and session calculi [17, 18]. In [2] and [3], Barbanera et al. investigated the notions of compliance and sub-behaviour for contracts with checkpoints. While rollbacks are forgetful in [2], in [3] they are used as a strategy to achieve compliance: in this case, after a rollback a process cannot engage again in the previously explored branch, presumably unsuccessful.

Our work is most closely related to the recent proposals [33, 34, 23, 22, 24, 28, 26, 14]. Tiezzi and Yoshida [33] use tags and memories to allow full reversibility of binary sessions with delegation. In [34], two forms of reversibility are considered: either a session is completely reversed in a single backward step, or any intermediate state is restored. Mezzina and Pérez [23, 22] use monitors as memories for reversing binary sessions. A key novelty of this work is the use of session types with present and past. In [24], this approach is generalised to multiparty sessions, asynchronous higher-order communications, and decoupled rollbacks. In [28], Neykova and Yoshida provide an algorithm to analyse and extract causal dependencies from a given multiparty global type, and use it to ensure that communicating processes are safely recovered from consistent states in the presence of a failure. In [26], Mezzina and Tuosto propose a semantic control of reversibility: a computation along a branch is reversed according to the guards on the current configuration. A feature of [26] is that inputs are potentially irreversible actions, unless they appear within a loop. Finally, our paper builds on [14] and improves it in several respects: it has a more liberal syntax for types and processes, gives a more compact representation of past communications and implements a fine-tuned strategy for backward moves, geared towards achieving compliance.
It should be noted that in all the above-mentioned proposals, the syntax of global types (if any) does not include recursion, parallel composition nor sequential composition. Thus, some of our examples cannot be expressed in these models. On the other hand, the context-free session types of [32], recently proposed to capture the type-safe serialisation of recursive data types, include sequential composition and recursion. Our session types generalise them by offering in addition parallel composition, and by handling multiparty sessions.

The properties of our calculus are standard for session types, but their proofs require some ingenuity due to the generality of our syntax and to the specificity of our rollback mechanism. A limitation of our work is that the starting points of rollbacks are statically determined. By contrast, these points are determined dynamically in [26], offering a more realistic solution. We plan to introduce similar runtime conditions for rollback in our calculus.

Since our session types and global types are “truly concurrent”, we would also like to study their interpretation into some non-interleaving semantic model. A natural candidate that comes to mind is the model of Event Structures, for which reversible variants have already been proposed [30, 16]. We plan to explore a reversible variant of Flow Event Structures [5], a model that has already been used to interpret CCS processes with past in [6].

Acknowledgements. We are grateful to the anonymous reviewers for their useful suggestions, which led to substantial improvements. We also thank Claudio Antares Mezzina for pointing out a technical problem in an earlier version of the paper.

References


Concurrent Reversible Sessions


Unbounded Product-Form Petri Nets∗†

Patricia Bouyer¹, Serge Haddad², and Vincent Jugé³

1 LSV, CNRS, ENS Paris-Saclay, Université Paris-Saclay, France
2 LSV, CNRS, ENS Paris-Saclay, Inria, Université Paris-Saclay, France
3 LSV, CNRS, ENS Paris-Saclay, Université Paris-Saclay, France

Abstract
Computing steady-state distributions in infinite-state stochastic systems is in general a very difficult task. Product-form Petri nets are those Petri nets for which the steady-state distribution can be described as a natural product corresponding, up to a normalising constant, to an exponentiation of the markings. However, even though some classes of nets are known to have a product-form distribution, computing the normalising constant can be hard. The class of (closed) Π³-nets has been proposed in an earlier work, for which it is shown that one can compute the steady-state distribution efficiently. However these nets are bounded. In this paper, we generalise queuing Markovian networks and closed Π³-nets to obtain the class of open Π³-nets, that generate infinite-state systems. We show interesting properties of these nets: (1) we prove that liveness can be decided in polynomial time, and that reachability in live Π³-nets can be decided in polynomial time; (2) we show that we can decide ergodicity of such nets in polynomial time as well; (3) we provide a pseudo-polynomial time algorithm to compute the normalising constant.

1998 ACM Subject Classification D.2.2 Petri Nets

Keywords and phrases Performance evaluation, infinite-state systems, Petri nets, steady-state distribution

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2017.31

1 Introduction
Quantitative analysis of stochastic infinite-state systems. Performance measures of stochastic systems can be roughly classified in two categories: those related to the transient behaviour as expressed by a temporal logic formula and those related to the long-run behaviour whose main measure is the steady-state distribution (when it exists).

There are different relevant questions concerning the steady-state distribution of infinite-state systems: (1) given a state and a threshold, one can ask whether the steady-state probability of this state is above (or below) the threshold; (2) given a state, one can compute the steady-state probability of this state, either in an exact way, or in an accurate approximate way; and (3) one can give a symbolic representation of both the set of reachable states and its associated distribution (or an accurate approximation thereof).

Clearly the last question is the most difficult one, and the first breakthrough in that direction has been obtained in the framework of open queuing Markovian networks: in those systems, the measure of a state is obtained as the product of terms, where each term is related to the parameters of a queue (service and visit rate) and the number of clients in the queue [13]. In order to get a probability distribution over the set of reachable states,
this product is normalised by a constant, whose computation is easy when the service rates of the queues do not depend on the number of clients. This work has been adapted to closed networks, and the main contribution in [9] consists in computing the normalising constant without enumerating the (finite) reachability set, leading to an algorithm which runs in polynomial-time w.r.t. the size of the network and the number of clients. Later, Markov chains generated by a stochastic Petri net with a single unbounded place (that is, quasi-birth death processes) have been investigated [8], and an algorithm which approximates up to arbitrary precision the steady-state distribution has been proposed; however the complexity of the algorithm is very high, since it requires the computation of the finite reachability sets of some subnets, whose size may be non primitive recursive. More recently, the abstract framework of (infinite-state) Markov chains with a finite eager attractor (e.g. probabilistic lossy channel systems) has been used to develop an algorithm which approximates up to arbitrary precision the steady-state distribution as well [1], but there is no complexity bound for the algorithm.

Product-form Petri nets. While queuing networks are very interesting since they allow for an explicit representation of the steady-state distribution, they lack two important features available in Petri nets [19, 20], which are very relevant for modelling concurrent systems: resource competition and process synchronisation. So very soon researchers have tried to get the best of the two formalisms and they have defined subclasses of Petri nets for which one can establish product-form steady-state distributions. Historically, solutions have been based on purely behavioural properties (i.e. by an analysis of the reachability graph) like in [15], and then progressively have moved to more and more structural characterisations [16, 6]. Building on the work of [6], [11] has established the first purely structural condition for which a product-form steady-state distribution exists, and designed a polynomial-time algorithm to check for the condition (see also [17] for an alternative characterisation). These nets are called $\Pi^2$-nets. However the computation of the normalising constant remains a difficult open issue since a naive approach in the case of finite-state $\Pi^2$-nets would require to enumerate the potentially huge reachability state-space. Furthermore, the lower bounds shown in [10] for behavioural properties of $\Pi^2$-nets strongly suggest that the computation of the normalising constant can probably not be done in an efficient way. In [6, 23], the authors introduce semantical classes of product-form Petri nets for which this constant is computed in pseudo-polynomial time. However their approach suffers two important drawbacks: (1) checking whether a net fulfills this condition as least as hard as the reachability problem, and (2) the only syntactical class for which this condition is fulfilled boils down to queuing networks.

To overcome this problem, the model of $\Pi^3$-nets is defined in [10] as a subclass of $\Pi^2$-nets obtained by structuring the synchronisation between concurrent activity flows in layers. This model strictly generalises closed product-form queuing networks (in which there is a single activity flow). Two interesting properties of those nets is that liveness for $\Pi^3$-nets and reachability for live $\Pi^3$-nets can both be checked in polynomial time. Furthermore, from a quantitative point-of-view, the normalising constant of the steady-state distribution can be efficiently computed using an elaborated dynamic programming algorithm.

Product-form Petri nets have been applied for the specification and analysis of complex systems. From a modelling point-of-view, compositional approaches have been proposed [18, 2] as well as hierarchical ones [12]. Application fields have also been identified, for instance, hardware design and more particularly RAID storage [12], or software architectures [3].
Our contributions. Unfortunately $\Pi^3$-nets generate finite-state systems. Here we address this problem by introducing and studying open $\Pi^3$-nets. Informally, an open $\Pi^3$-net has a main activity flow, which roughly corresponds to an open queuing network, and has other activity flows which are structured as in (standard, or closed) $\Pi^3$-nets. More precisely, in the case of a single activity flow this model is exactly equivalent to an open queuing network, but the general model is enriched with other activity flows, raising difficult computation issues. In particular, several places may in general be unbounded. Open $\Pi^3$-nets are particularly appropriate when modelling, in an open environment, protocols and softwares designed in layers. In addition, they allow to specify dynamical management of resources where processes may produce and consume them with no a priori upper bound on their number. Our results on open $\Pi^3$-nets can be summarised as follows:

- We first establish that the liveness problem can be solved in polynomial time, and that the boundedness as well as the reachability problem in live nets can also be solved in polynomial time. On the other side, we show that the unboundedness, the reachability and even the covering problem become NP-hard without the liveness assumption.

- Contrary to the case of closed $\Pi^3$-nets, open $\Pi^3$-nets may not be ergodic (that is, there may not exist a steady-state distribution). We design a polynomial-time algorithm to decide ergodicity of an open $\Pi^3$-net.

- Our main contribution is the computation of the normalising constant for ergodic live $\Pi^3$-nets. Our procedure combines symbolic computations and dynamic programming. The time complexity of the algorithm is polynomial w.r.t. the size of the structure of the net and the maximal value of integers occurring in the description of the net (thus pseudo-polynomial). As a side result, we improve the complexity for computing the normalising constant of closed $\Pi^3$-nets that was given in [10] (the complexity was the same, but was assuming that the number of activity flows is a constant).

In Section 2, we introduce and illustrate product-form nets, and recall previous results. In Section 3, we focus on qualitative behavioural properties, while quantitative analysis is developed in Section 4. All proofs are postponed to the full version of this article.

2 Product-form Petri nets

Notations. Let $A$ be a matrix over $I \times J$, one denotes $A(i,j)$ the item whose row index is $i$ and column index is $j$. When $I$ and $J$ are disjoint, $W(k)$ denotes the row (resp. column) vector indexed by $k$ when $k \in I$ (resp. $k \in J$). Given a real vector $v$ indexed by $I$ its norm, denoted $\|v\|$, is defined by $\|v\| = \sum_{i \in I} |v(i)|$. Sometimes, one writes $v_I$ for $v(i)$. Given two vectors $v, w$ indexed by $I$ their scalar product denoted $v \cdot w$ is defined by $\sum_{i \in I} v_i \cdot w_i$. Finally, if $v$ is a vector over $I$, we define its support as $\text{Supp}(v) = \{i \in I \mid v(i) \neq 0\}$.

We briefly recall Petri nets and stochastic Petri nets. The state of a Petri net, called a marking is defined by the number of tokens contained in every place. A Petri net models concurrent activities by transitions whose enabling requires tokens to be consumed in some places and then tokens to be produced in some places.

Definition 1 (Petri net). A Petri net is a tuple $\mathcal{N} = (P, T, W^-, W^+)$ where:

- $P$ is a finite set of places;
- $T$ is a finite set of transitions, disjoint from $P$;
- $W^-$ and $W^+$ are $P \times T$ matrices with coefficients in $\mathbb{N}$.

$W^-$ (resp. $W^+$) is called the backward (resp. forward) incidence matrix, $W^-(p,t)$ (resp. $W^+(p,t)$) specifies the number of tokens consumed (resp. produced) in place $p$ by the firing of transition $t$, and $W^-(t)$ (resp. $W^+(t)$) is the $t$-th column of $W^-$ (resp. $W^+$). One
assumes that for all \( t \in T \), \( W^{-}(t) \neq W^{+}(t) \) (i.e. no useless transition) and for all \( t' \neq t \), either \( W^{-}(t) \neq W^{-}(t') \) or \( W^{+}(t) \neq W^{+}(t') \) (i.e. no duplicated transition); this will not affect our results.

A marking of \( \mathcal{N} \) is a vector of \( \mathbb{N}^{P} \); in the sequel we will often see \( m \) as a multiset (\( m(p) \) is then the number of occurrences of \( p \)), or as a symbolic sum \( \sum_{p \in P} m(p) p \). The symbolic sum \( \sum_{p \in P} p \) will be more concisely written \( P' \). Transition \( t \) is enabled by marking \( m \in \mathbb{N}^{P} \) if for all \( p \in P \), \( m(p) > W^{-}(p, t) \). When enabled, its firing leads to the marking \( m' \) defined by: for all \( p \in P \), \( m'(p) = m(p) - W^{-}(p, t) + W^{+}(p, t) \). This firing is denoted by \( m \xrightarrow{t} m' \). The incidence matrix \( W = W^{+} - W^{-} \) allows one to rewrite the marking evolution as \( m' = m + W(t) \) if \( m \geq W^{-}(t) \). Given an initial marking \( m_{0} \in \mathbb{N}^{P} \), the reachability set \( \mathcal{R}_{\mathcal{N}}(m_{0}) \) is the smallest set containing \( m_{0} \) and closed under the firing relation. When no confusion is possible one denotes it more concisely by \( \mathcal{R}(m_{0}) \). Later, if \( m \in \mathcal{R}(m_{0}) \), we may also write \( m_{0} \rightarrow^{*} m \). We will call \( (\mathcal{N}, m_{0}) \) a marked Petri net.

From a qualitative point of view, one is interested by several standard relevant properties including reachability. Liveness means that the modelled system never loses its capacity: for all \( t \in T \) and \( m \in \mathcal{R}(m_{0}) \), there exists \( m' \in \mathcal{R}(m) \) such that \( t \) is enabled in \( m' \). Boundedness means that the modelled system is a finite-state system: there exists \( B \in \mathbb{N} \) such that for all \( m \in \mathcal{R}(m_{0}) \), \( \|m\| \leq B \). While decidable, these properties are costly to check: (1) Reachability is \( \text{EXPSpace-hard} \) in general and \( \text{PSPACE-complete} \) for 1-bounded nets [7], (2) using results of [21] liveness has the same complexity, and (3) boundedness is \( \text{EXPSpace-complete} \) [22]. Furthermore there is a family of bounded nets \( \{\mathcal{N}_{n}\}_{n \in \mathbb{N}} \) whose size is polynomial in \( n \) such that the size of their reachability set is lower bounded by some Ackermann function [14].

**Example 2.** An example of marked Petri net is given on Figure 1. Petri nets are represented as bipartite graphs where places are circles containing their initial number of tokens and transitions are rectangles. When \( W^{-}(p, t) > 0 \) (resp. \( W^{+}(p, t) > 0 \)) there is an edge from \( p \) (resp. \( t \)) to \( t \) (resp. \( p \)) labelled by \( W^{-}(p, t) \) (resp. \( W^{+}(p, t) \)). This label is called the weight of this edge, and is omitted when its value is equal to one. For sake of readability, one merges edges \( p \xrightarrow{W^{-}(p, t)} t \) and \( t \xrightarrow{W^{+}(p, t)} p \) when \( W^{-}(p, t) = W^{+}(p, t) \) leading to a pseudo-edge with two arrows, as in the case of \((r_{0}, t_{5})\).

The net of Figure 1 is not live. Indeed \( t_{0}, t_{1}, t_{2}, t_{3} \) and \( t_{4} \) will never be enabled due to the absence of tokens in \( p_{0}, p_{1}, p_{2}, \text{pext} \). Suppose that one deletes the place \( \text{pext} \) and its input.
and output edges. Consider the firing sequence $q_3 + r_0 \xrightarrow{t_0} q_2 + r_0 \xrightarrow{t_0} q_1 + r_0 \xrightarrow{t_0} p_0 + r_0 \xrightarrow{t_1} p_2 + 3q_3 + r_0$. Since the marking $p_2 + 3q_3 + r_0$ is (componentwise) larger than the initial marking $q_3 + r_0$, we can iterate this sequence and generate markings with an arbitrarily large number of tokens in $p_2$ and $q_3$; this new net is unbounded. Applying techniques that we will develop in this paper (Section 3), we will realize that this new net is actually live.

▶ **Definition 3** (Stochastic Petri net). A **stochastic Petri net** (SPN) is a pair $(\mathcal{N}, \lambda)$ where:

- $\mathcal{N} = (P, T, W^-, W^+)$ is a Petri net;
- $\lambda$ is a mapping from $T$ to $\mathbb{R}_{>0}$.

A marked stochastic Petri net $(\mathcal{N}, \lambda, m_0)$ is a stochastic Petri net equipped with an initial marking. In a marked stochastic Petri net, when becoming enabled a transition triggers a random delay according to an exponential distribution with (firing) rate $\lambda(t)$. When several transitions are enabled, a race between them occurs. Accordingly, given some initial marking $m_0$, the stochastic process underlying a SPN is a continuous time Markov chain (CTMC) whose (possibly infinite) set of states is $R(m_0)$ and such that the rate $\mathbf{Q}(m, m')$ of a transition from some $m$ to some $m' \neq m$ is equal to $\sum_{t, m \rightarrow m'} \lambda(t)$ (as usual $\mathbf{Q}(m, m) = -\sum_{m' \neq m} \mathbf{Q}(m, m')$). Matrix $\mathbf{Q}$ is called the infinitesimal generator of the CTMC (see [5] for more details).

From a quantitative point of view, one may be interested in studying the long-run behaviour of the net and in particular in deciding whether there exists a steady-state distribution and in computing it in the positive case. When the underlying graph of the CTMC is strongly connected (i.e. an irreducible Markov chain) it amounts to deciding whether there exists a non-zero distribution $\pi$ over $R(m_0)$ such that $\pi \cdot \mathbf{Q} = 0$.

It is in general non-trivial to decide whether there exists a steady-state distribution, and even when such a distribution exists, given some state it is hard to compute its steady-state probability (see the introduction). Furthermore, even when the net is bounded, the size of the reachability set may prevent any feasible computation of $\pi$.

Thus one looks for subclasses of nets where the steady-state distribution $\pi$ can be computed more easily and in particular when $\pi$ has a product-form, that is: there exist a constant vector $\mu \in \mathbb{R}^P_+$ only depending on $\mathcal{N}$ such that for all $m \in R(m_0)$, $\pi(m) = G \cdot \prod_{p \in P} \mu_p^{m(p)}$ where $G = \left( \sum_{m \in R(m_0)} \prod_{p \in P} \mu_p^{m(p)} \right)^{-1}$ is the so-called normalising constant [9].

The most general known class of nets admitting a structural product-form distribution is the class of $\Omega^2$-nets [11]. It is based on two key ingredients: bags and witnesses. A bag is a multiset of tokens that is consumed or produced by some transition. Considering a bag as a whole, one defines the bag graph whose vertices are bags and, given a transition $t$, an edge goes from the bag consumed by $t$ to the bag produced by $t$. Observe that there are at most $2|T|$ vertices and exactly $|T|$ edges. This alternative representation of a net via the bag graph does not lose any information: from the bag graph, one can recover the original net. Formally:

▶ **Definition 4** (Bag graph of a Petri net). Let $\mathcal{N} = (P, T, W^-, W^+)$ be a Petri net. Then its bag graph is a labelled graph $G_{\mathcal{N}} = (V_{\mathcal{N}}, E_{\mathcal{N}})$ defined by:

- $V_{\mathcal{N}} = \{W^-(t), W^+(t) \mid t \in T\}$ the finite set of bags;
- $E_{\mathcal{N}} = \{W^-(t) \xrightarrow{t} W^+(t) \mid t \in T\}$.

▶ **Example 5.** The bag graph of the net of Figure 1 is described in Figure 2. The bag is written inside the vertex (the external label of the vertices will be explained later). Observe that this graph has three connected components, both of them being strongly connected.
We now turn to the notion of witness. A queuing network models a single activity flow where activities are modelled by queues and clients leave their current queue when served and enter a new one depending on a routing probability. In $\Pi^2$-nets there are several activity flows, one per component of the bag graph. So one wants to witness production and consumption of every bag $b$ by the transition firings. In order to witness it, one looks for a linear combination of the places $\text{wit}$ such that for every firing of a transition $t$ that produces (resp. consumes) the bag $b$, for every marking $m$, $m \cdot \text{wit}$ is increased (resp. decreased) by one unit, and such that all other transition firings let $m \cdot \text{wit}$ invariant.

\begin{itemize}
  \item \textbf{Definition 6 (Witness of a bag).} Let $\mathcal{N} = (P, T, W^-, W^+)$ be a a Petri net, $b \in V_{\mathcal{N}}$ and $\text{wit} \in \mathbb{Q}^P$. Then $\text{wit}$ is a witness of $b$ if:

  \begin{align*}
  \text{wit} \cdot W(t) &= -1 \quad \text{if } W^-(t) = b; \\
  \text{wit} \cdot W(t) &= 1 \quad \text{if } W^+(t) = b; \\
  \text{wit} \cdot W(t) &= 0 \quad \text{otherwise.}
  \end{align*}

  \item \textbf{Example 7.} All bags of the net of Figure 1 have (non unique) witnesses. We have depicted them close to their vertices in Figure 2. For instance, consider the bag $b = q_1 + r_0$: transition $t_6$ produces $b$ while transition $t_7$ consumes it. Let us check that $w = q_1 - p_{\text{ext}}$ is a witness of $b$. $t_6$ produces a token in $q_1$ and the marking of $p_{\text{ext}}$ is unchanged. $t_7$ consumes a token in $q_1$ and the marking of $p_{\text{ext}}$ is unchanged. The other transitions that change the marking of $q_1$ and $p_{\text{ext}}$ are $t_1$ and $t_3$. However since they simultaneously produce or consume a token in both places, $m \cdot w$ is unchanged ($m$ is the current marking).

The definition of $\Pi^2$-nets relies on structural properties of the net and on the existence of witnesses. Every connected component of the graph bag will represent an activity flow of some set of processes where every activity (i.e. a bag) has a witness.

\begin{itemize}
  \item \textbf{Definition 8 ($\Pi^2$-net).} Let $\mathcal{N}$ be a Petri net. Then $\mathcal{N}$ is a $\Pi^2$-net if:

  \begin{itemize}
    \item every connected component of $G_{\mathcal{N}}$ is strongly connected;
    \item every bag $b$ of $V_{\mathcal{N}}$ admits a witness (denoted $\text{wit}_b$).
  \end{itemize}

  \end{itemize}

Observe that the first condition called weak reversibility ensures that the reachability graph is strongly connected since the firing of any transition $t$ can be "withdrawn" by the firing of transitions occurring along a path from $W^+(t)$ to $W^-(t)$ in the bag graph. The complexity of reachability in weakly reversible nets is still high: EXPSPACE-complete [4].

The next theorem shows the interest of $\Pi^2$-nets. Let us define $\lambda(b)$ the firing rate of a bag $b$ by $\lambda(b) = \sum_{t|W^-(t)=b} \lambda(t)$ and the choice probability $pr_t$ of transition $t$ by $pr_t = \frac{\lambda(t)}{\lambda(W^-(t))}$.
The routing matrix $P$ of bags is the stochastic matrix indexed by bags such that for all $t$, $P(W^+(t), W^-(t)) = pr_t$ and $P(b, b') = 0$ otherwise. Consider $\text{vis}$ some positive solution of $\text{vis} \cdot P = \text{vis}$. Since $P$ is a stochastic matrix such a vector always exists but is not unique in general; however given $b, b'$ two bags of the same connected component $\text{vis}(b') / \text{vis}(b)$ measures the ratio between visits of $b'$ and $b$ in the discrete time Markov chain induced by $P$.

**Theorem 9** (**[11]**). Let $(N', \lambda, m_0)$ be a marked stochastic $\Pi^2$-net. Then defining for all $m \in \mathcal{R}(m_0)$, $v(m) = \prod_{b \in V_N} \left( \frac{\text{vis}(b)}{\text{vis}(b')} \right)^{m \cdot \text{w}(b)}$, the next assertions hold:

- $v \cdot Q = 0$;
- $(N', \lambda, m_0)$ is ergodic iff $\|v\| < \infty$
- when ergodic, the associated Markov chain admits $\|v\|^{-1}v$ as steady-state distribution.

Let us discuss the computational complexity of the product-form of the previous theorem. First deciding whether a net is a $\Pi^2$-net is straightforwardly performed in polynomial time. The computation of the visit ratios, the witnesses and the rate of bags can also be done in polynomial time. So computing an item of vector $v$ is easy. However without additional restriction on the nets the normalising constant $\|v\|^{-1}$ requires to enumerate all items of $\mathcal{R}_N(m_0)$, which can be prohibitive.

So in [10], the authors introduce $\Pi^3$-net, a subclass of $\Pi^2$-net which still strictly generalises closed queueing networks, obtained by structuring the activity flows of the net represented by the components of the bag graph. First there is a bijection between places and bags such that the input (resp. output) transitions of the bag produce (consume) one token of this place. The other places occurring in the bag may be ordered as strongly connected components of the graph may be ordered as $N$ layers such that the resources of a bag occurring in layer $i$ correspond to places associated with bags of layer $i-1$ (for $i > 1$) and more precisely to those with maximal potential. Informally a token in such a place means that it is a resource available for the upper layer.

**Definition 10** ($\Pi^3$-net). Let $N$ be a net. Then $N$ is an $N$-closed $\Pi^3$-net if:

- There is a bijection between $P$ and $V_N$. Denoting $b_p$ the bag associated with place $p$ (and $p_0$ the place associated with bag $b$), we have $b_p(p_0) = 1$.
- The potential of a place $\text{pot}(p)$ is equal to $\|b_p\| - 1$.
- $V_N$ is partitioned into $N$ strongly connected components $V_1, \ldots, V_N$. One denotes:
  - $P_1 = \{p_0 \mid b \in V_1\}$ and $P_1^{\text{max}} = \arg\max(\text{pot}(p) \mid p \in P_1)$. By convention, $P_0^{\text{max}} = P_0 = \emptyset$.
  - For all $b \in V_i$ and $p \in P \setminus \{p_0\}$, $b(p) > 0$ implies $p \in P_i^{\text{max}}$.

A net is an $N$-open $\Pi^3$-net if it is obtained by deleting some place $p_{\text{ext}} \in P_N$ (and its input/output edges) from an $N$-closed $\Pi^3$-net.

Given an open $\Pi^3$-net $\overline{N}$, $\overline{N}$ denotes the closed net based on which $N$ has been defined. For every $1 \leq i \leq N$, we will later write $P_i^{\text{max}}$ for the set $P_1 \setminus P_i^{\text{max}}$, and $T_i$ for the set of transitions $t$ such that $W^-(t) \in V_i$. The next proposition establishes that $\Pi^3$-nets are product-form Petri nets.

**Proposition 11.** Let $N$ be a (closed or open) $\Pi^3$-net. Then $N$ is a $\Pi^2$-net.

**Example 12.** The net of Figure 1 is a 3-closed $\Pi^3$-net. We have used different colors for the layers: $P_3 = \{p_0, p_1, P_2, p_{\text{ext}}\}$ (in red), $P_2 = \{q_0, q_1, q_2, q_3\}$ (in green) and $P_1 = \{r_0, r_1\}$ (in blue). The place $q_0$ (resp. $q_1, q_2, q_3$) is associated with the bag $q_0$ (resp. $q_1+r_0, q_2+r_0, q_3+r_0$) and has potential 0 (resp. 1, 1, 1). Thus $P_2^{\text{max}} = \{q_1, q_2, q_3\}$ and indeed $q_0$ does not occur in bags of layer 3. In order to highlight the use of resources, edges between places of $P_i$ and transitions of $T_{i+1}$ are depicted in gray.
The next theorem shows the interest of closed $\Pi^3$-nets.

\textbf{Theorem 13 ([10])}. Let $(\mathcal{N}, \lambda, m_0)$ be a $N$-closed $\Pi^3$-net. Then:

- $(\mathcal{N}, m_0)$ is bounded.
- One can decide whether $(\mathcal{N}, m_0)$ is live in polynomial time.
- When $(\mathcal{N}, m_0)$ is live, one can decide in polynomial time, given $m$, whether $m \in \mathcal{R}_\mathcal{N}(m_0)$.
- For any fixed $N$, when $(\mathcal{N}, m_0)$ is live, one can compute the normalising constant of the steady-state distribution (i.e. $\|\mathbf{v}\|^{-1}$ in Theorem 9) in polynomial time with respect to $|\mathcal{P}|$, $|\mathcal{T}|$, the maximal weight of the net’s edges, and $\|m_0\|$ (thus in pseudo-polynomial time w.r.t. the size of $m_0$).

The above results do not apply to infinite-state systems and in particular to the systems generated by open $\Pi^3$-nets. In addition, the polynomial-time complexity for computing the normalising constant requires to fix $N$. We address these issues in the next sections.

\section{Qualitative analysis}

In this section we first give a simple characterisation of the liveness property in a marked $\Pi^3$-net. We then fully characterise the set of reachable markings in a live marked $\Pi^3$-net. These characterisations give polynomial-time algorithms for deciding liveness of a marked $\Pi^3$-net, and the boundedness property of a live marked $\Pi^3$-net. We end the section with a coNP-hardness result for the boundedness property of a marked $\Pi^3$-net, when it is not live.

For the rest of this section, we assume $\mathcal{N} = (\mathcal{P}, \mathcal{T}, W^-, W^+)$ is an open or closed $\Pi^3$-net with $N$ layers. We further use the notations of Definition 10. In particular, if $\mathcal{N}$ is open, then we write $p_{\text{ext}}$ for the place which has been removed (and we call it virtual). We therefore set $P^*_N = P_N \cup \{p_{\text{ext}}\}$ if the net is open and $P^*_N = P_N$ otherwise; For every $1 \leq i \leq N - 1$ we define $P^*_i = P_i$; And we set $P^*_N = \bigcup_{i=1}^N P^*_i$.

\subsection{Liveness analysis}

We give a simple characterisation of the liveness property through a dependence between the number of tokens at some layer and potentials of places activated on the next layer. More precisely, for every $1 \leq i \leq N - 1$, $\text{Live}_i$ is defined as the set of markings $m$ such that:

$$m \cdot P_i \geq \min\{\text{pot}(p) \mid p \in P^*_i+1 \text{ and } (m(p) > 0 \text{ or } p = p_{\text{ext}})\}.$$  

Note that $p = p_{\text{ext}}$ can only happen when $\mathcal{N}$ is open and $i = N - 1$. We additionally define $\text{Live}_N$ as the set of markings $m$ such that $m \cdot P_N > 0$ if $\mathcal{N}$ is closed, and as the set of all markings if $\mathcal{N}$ is open.

For every $1 \leq i \leq N$ (resp. $1 \leq i \leq N - 1$), we define $\text{POT}_i = \max\{\text{pot}(p) \mid p \in P_i\}$ when $\mathcal{N}$ is closed (resp. open). When $\mathcal{N}$ is open, we write $\text{POT}_N = \text{pot}(p_{\text{ext}})$. Given a marking $m$, when no place p fulfills $p \in P^*_i+1$ and $(m(p) > 0 \text{ or } p = p_{\text{ext}})$, the minimum is equal to $\text{POT}_{i+1}$. Thus given a marking $m$, the condition $m \in \text{Live}_i$ for $i < n$ only depends on the values of $m(p)$ for $p \in P_i \cup P^*_{i+1}$.

The intuition behind condition $\text{Live}_i$ is the following: transitions in $\bigcup_{j \leq i} T_j$ cannot create new tokens on layer $i$ (layer $i$ behaves like a state machine, and smaller layers do not change the number of tokens in that layer); therefore, to activate a transition of $T_{i+1}$ out of some marked place $p \in P^*_{i+1}$, it must be the case that enough tokens are already present on layer $i$; hence there should be at least as many tokens in layer $i$ as the minimal potential of a marked place in layer $i + 1$. When $\mathcal{N}$ is open, the virtual place $p_{\text{ext}}$ behaves like a source
of tokens, hence it is somehow always “marked”; this is why it is taken into account in the right part of \( \text{Live}_i \). The following characterisation was already stated in [10] in the restricted case of closed nets.

**Theorem 14.** A marking \( m \) is live if and only if for every \( 1 \leq i \leq N \), \( m \in \text{Live}_i \).

**Example 15.** Building on the marked Petri net of Figure 1, the marking \( q_3 + r_0 \) is live when the net is open but not live if the net is closed. Indeed, transitions of the two first layers can trivially be activated from \( q_3 + r_0 \) (hence by weak-reversibility from every reachable marking). We see that in the case of the closed net, transitions of layer 3 cannot be activated (no fresh token can be produced on that layer). On the contrary, in the case of the open net, the token in \( q_3 \) can be moved to \( q_1 \), which will activate transition \( t_3 \); from there, all transitions of layer 3 will be eventually activated.

As a consequence of the characterisation of Theorem 14, we get:

**Corollary 16.** We can decide the liveness of a marked \( \Pi^3 \)-net in polynomial time.

### 3.2 Reachable markings

We will now give a characterisation of the set of reachable markings \( R_N(m_0) \) when \( m_0 \) is live. We will first give linear invariants of the net: those are vectors in the left kernel of \( W \) (or \( P \)-flows). The name “invariants” comes from the fact that they will allow to infer real invariants satisfied by the reachable markings. Furthermore, for every \( 1 \leq i \leq N \), for every \( p \in P_i \), we define \( \text{cin}(p) = \text{POT}_i - \text{pot}(p) \). Except when \( N \) is open and \( i = N \), the \( \text{cin} \) value of a place is nonnegative.

**Proposition 17.** The following vectors are linear invariants of \( N \):

- for every \( 1 \leq i \leq N - 1 \), \( v^{(i)} = \sum_{p \in P_i} p + \sum_{p \in P_{i+1}} \text{cin}(p) \);  
- if \( N \) is closed, \( v^{(N)} = \sum_{p \in P_N} p \).

First observe that for \( i < N - 1 \), \( \text{Supp}(v^{(i)}) = P_i \cup P_{i+1}^{\text{max}} \). Thus given a marking \( m \), only firing of transitions \( t \in T_i \cup T_{i+1} \) could change \( m \cdot v^{(i)} \). This is not the case of a transition \( t \in T_i \) since it moves a token from a place of \( P_i \) to another one.

To give an intuition why transitions in \( T_{i+1} \) do not change \( m \cdot v^{(i)} \), we consider part of the closed net (that is, \( P_{\text{ext}} \) is a real place) of Figure 1 depicted on the right, where numbers close to place names are potential values. We focus on transition \( t_1 \) and explain why \( m \cdot v^{(2)} \) is unchanged by its firing. The impact of transition \( t_1 \) is to decrease the sum \( \sum_{p \in P_3} p \) by 2; due to the weights of places of \( P_3 \) in \( v^{(2)} \), place \( p_2 \) counts as 0 and place \( p_{\text{ext}} \) counts as +2. This intuition extends into a formal proof.

For every \( 1 \leq i \leq N - 1 \), we set \( C_i^{m_0} = m_0 \cdot v^{(i)} \), and if \( N \) is closed, we set \( C_N^{m_0} = m_0 \cdot v^{(N)} \). For every \( 1 \leq i \leq N - 1 \), we define \( \text{Inv}_i(m_0) \) as the set of markings \( m \) such that \( m \cdot v^{(i)} = C_i^{m_0} \), and if \( N \) is closed, we define \( \text{Inv}_N(m_0) \) as the set of markings \( m \) such that \( m \cdot v^{(N)} = C_N^{m_0} \). For uniformity, if \( N \) is open, we define \( \text{Inv}_N(m_0) \) as the set of all markings. As a consequence of Proposition 17, we get:

**Corollary 18.** \( R_N(m_0) \subseteq \bigcap_{i=1}^N \text{Inv}_i(m_0) \).
More complex invariants were given in [10] for closed \( \Pi^3 \)-nets. The advantage of the above invariants is that each of them only involves two neighbouring layers. This will have a huge impact on various complexities, and will allow the development of our methods for quantitative analysis.

\textbf{Example 19.} Going back to the Petri net of Figure 1, with initial marking \( m_0 = q_3 + r_0 \). We first consider the closed net. Then: \( \text{POT}_3 = 2 \) and \( \text{POT}_2 = 1 \). Therefore:

\[
\begin{aligned}
\text{Inv}_3(m_0) &= \{ m | \sum_{i=0}^{2} m(p_i) + m(p_{\text{ext}}) = 0 \} \\
\text{Inv}_2(m_0) &= \{ m | \sum_{i=0}^{3} m(q_i) + 3m(p_0) + 2m(p_1) + 2m(p_{\text{ext}}) = m_0(q_3) = 1 \} \\
\text{Inv}_1(m_0) &= \{ m | m(r_0) + m(r_1) + m(q_0) = m_0(r_0) = 1 \}
\end{aligned}
\]

We now turn to the open net, obtained by deleting \( p_{\text{ext}} \). Definition of \( \text{POT}_3 \) differs from the previous case: \( \text{POT}_3 = \text{pot}(p_{\text{ext}}) = 1 \) and \( \text{POT}_2 = 1 \). Therefore:

\[
\begin{aligned}
\text{Inv}_3(m_0) &= \mathbb{N}^P \\
\text{Inv}_2(m_0) &= \{ m | \sum_{i=0}^{3} m(q_i) + m(p_0) - 2m(p_2) = m_0(q_3) = 1 \} \\
\text{Inv}_1(m_0) &= \{ m | m(r_0) + m(r_1) + m(q_0) = m_0(r_0) = 1 \}
\end{aligned}
\]

The invariants of Corollary 18 do not fully characterise the set of reachable markings, since they do not take into account the enabling conditions of the transitions. However, they will be very helpful for characterising the reachable markings when \( m_0 \) is live.

\textbf{Theorem 20.} Suppose that \( (N, m_0) \) is a live \( \Pi^3 \)-net. Then:

\[
\mathcal{R}_N(m_0) = \bigcap_{i=1}^{N} \text{Inv}_i(m_0) \cap \bigcap_{i=1}^{N} \text{Live}_i
\]

Thus reachability in live \( \Pi^3 \)-nets can be checked in polynomial time.

\textbf{Example 21.} In the open Petri net of Figure 1, with initial marking \( m_0 = q_3 + r_0 \), the sets \( \text{Live}_i \) are \( \text{Live}_1 = \{ m | m(r_0) + m(r_1) + m(q_0) \geq 1 \} \), \( \text{Live}_2 = \{ m | \sum_{i=0}^{3} m(q_i) + m(p_0) \geq 1 \} \) and \( \text{Live}_3 = \mathbb{N}^P \). Observing that \( \text{Inv}_i(m_0) \subseteq \text{Live}_i \) for \( 1 \leq i \leq 3 \), the net has reachability set

\[
\mathcal{R}_N(m_0) = \{ m | m(r_0) + m(r_1) + m(q_0) = 1 \text{ and } \sum_{i=0}^{3} m(q_i) + m(p_0) - 2m(p_2) = 1 \}.
\]

The idea of the proof when the net is closed is to show that, from every marking \( m \) satisfying the right handside condition in the theorem, one can reach a specific marking \( m_0^\ast \) (where, for every \( 1 \leq i \leq N \), \( C_i^{m_0^\ast} \) tokens are in one arbitrary place of \( P_i^{\text{max}} \)). Hence, given two markings \( m \) and \( m' \) satisfying the conditions, \( m \rightarrow^* m_0^\ast \) and \( m' \rightarrow^* m_0^\ast \), which implies by weak reversibility of the net: \( m \rightarrow^* m_0^\ast \rightarrow^* m' \). This is in particular the case from \( m_0 \); every marking satisfying the conditions is reachable from \( m_0 \). In the case of an open net, this is a bit more tricky, and a joint marking to every pair \( (m, m') \) of markings satisfying the conditions has to be chosen.

### 3.3 Boundedness analysis

As a consequence of the characterisation given in Theorem 20, we get:

\textbf{Corollary 22.} We can decide the boundedness of a live marked \( \Pi^3 \)-net in polynomial time.
Indeed, it can be shown that, if $\mathcal{N}$ is closed, then $\mathcal{N}$ is bounded, and that if $\mathcal{N}$ is open, then it is bounded if and only if $\cin(q) > 0$ for all $q \in P_N$ (that is, $p_{ext}$ has maximal potential, and no other place of $P_N$ has maximal potential). Furthermore, if $\mathcal{N}$ is bounded, then the overall number of tokens in the net is bounded by $\sum_{i=1}^{N-1} C_i^{m_0}$ (resp. $\sum_{i=1}^{N} C_i^{m_0}$) if the net is open (resp. closed).

The polynomial-time complexity of Corollary 22 is in contrast with the following hardness result, which can be obtained by a reduction from the independent-set problem.

**Proposition 23.** Deciding the boundedness of a marked $\Pi^3$-net which is not live is coNP-hard. The reachability (and even the coverability) problem is NP-hard.

### 4 Quantitative analysis

Contrary to closed $\Pi^3$-nets, open $\Pi^3$-nets may not be ergodic. In this section, we first give a simple characterisation of the ergodicity property for open $\Pi^3$-nets, which gives us a polynomial-time algorithm for deciding ergodicity. We then provide a polynomial-time algorithm for computing the steady-state distribution of ergodic (open and closed) $\Pi^3$-nets.

For the rest of this section, we assume that $(\mathcal{N}, \lambda, m_0)$ is a stochastic $\Pi^3$-net with $N$ layers, and that $m_0$ is live. Let $W$ be the maximal weight of the edges of $\mathcal{N}$. Then we assume that the constants $\mu_p = \prod_{b \in V_E} (\vis(b)/\lambda_b)^{\vis_p}$ have already been precomputed (in polynomial time with respect to $|P|$, $|T|$ and $\log(1 + W)$). \(^1\)

In what follows, and for every vector $\delta \in \mathbb{N}^P$, we denote by $\hat{v}(\delta)$ the product $\prod_{p \in P} \mu_p^{\delta(p)}$. Consequently, the vector $v$ mentioned in Theorem 9 is then defined by $v(m) = \hat{v}(m)$ for all markings $m \in \mathcal{R}(m_0)$, and its norm is $\|v\| = \sum_{m \in \mathcal{R}(m_0)} \hat{v}(m)$. Note that $v$ and $\hat{v}$ only differ by their domain. In addition, in what follows, and for every set $Z \subseteq P$, we simply denote by $\cin(Z)$ the formal sum $\sum_{p \in Z} \cin(p)p$.

#### 4.1 Ergodicity analysis

We assume here that $\mathcal{N}$ is open. We give a simple characterisation of the ergodicity property through a comparison of the constants $\mu_p$ for a limited number of places $p$. Those constraints express congestion situations that may arise; we show that they are sufficient. These places are the elements of the subset $Y$ of places that is defined by $Y = P_N \cup P_{N-1}^{max}$. In particular, as soon as the initial marking $m_0$ is live, then the ergodicity of the stochastic net $(\mathcal{N}, \lambda, m_0)$ does not depend on $m_0$.

According to Theorem 9, the net is ergodic if and only if the norm $\|v\| = \sum_{m \in \mathcal{R}(m_0)} \hat{v}(m)$ is finite. Hence, deciding ergodicity amounts to deciding the convergence of a sum. The following characterisation holds.

**Theorem 24.** Let $(\mathcal{N}, \lambda, m_0)$ be a live open stochastic $\Pi^3$-net with $N$ layers. This net is ergodic if and only if all of the following inequalities hold:

- for all places $p \in P_N$, if $\cin(p) = 0$, then $\mu_p < 1$;
- for all places $p, q \in P_N$, if $\cin(p) > 0 > \cin(q)$, then $\mu_p^{\cin(q)} \mu_q^{\cin(p)} < 1$;
- for all places $p \in P_{N-1}^{max}$ and $q \in P_N$, if $0 > \cin(q)$, then $\mu_p^{\cin(q)} \mu_q < 1$.

**Proof (sketch).** Let $F$ be the family formed of the vectors $p$ (for $p \in P_N$ such that $\cin(p) = 0$), $\cin(p)q - \cin(q)p$ (for $p, q \in P_N$ such that $\cin(p) > 0 > \cin(q)$) and $q - \cin(q)p$ (for

\(^1\) In the rest of this section, we will design polynomial-time procedures w.r.t. $W$, hence polynomial if it is encoded in unary and pseudo-polynomial if it is encoded in binary.
Unbounded Product-Form Petri Nets

Let \( p \in P^\text{max}_N \) and \( q \in P_N \) such that \( 0 > \text{cin}(q) \). Let also \( \mathcal{L} \) be the sublattice of \( \mathbb{N}^P \) generated by the vectors in \( \mathcal{F} \), and let \( \mathcal{G} \) be the finite subset of \( \mathbb{N}^P \) formed of those vectors whose entries are not greater than some adequately chosen constant \( \mathcal{G} \).

Since \( m_0 \) is live, Theorem 20 applies, which allows us to prove the inclusions \( \{m_0\} + \mathcal{L} \subseteq \mathcal{R}(m_0) \subseteq \mathcal{G} + \mathcal{L} \). Hence, the sum \( \sum_{m \in \mathcal{R}(m_0)} \hat{v}(m) \) is finite iff the sum \( \sum_{m \in \mathcal{L}} \hat{v}(m) \) is finite, i.e., iff each constant \( \hat{v}(\delta) \) is (strictly) smaller than 1, for \( \delta \in \mathcal{F} \).

**Example 25.** Going back to the open \( \Pi^3 \)-net of Figure 1, with any live initial marking \( m_0 \), we obtain the following necessary and sufficient conditions for being ergodic:

\[
\mu_{p_1} < 1, \quad \mu_{p_1}^2 \mu_{p_2} < 1, \quad \mu_{q_1}^2 \mu_{p_2} < 1, \quad \mu_{q_2}^2 \mu_{p_2} < 1.
\]

As a consequence of the characterisation of Theorem 24, we get:

**Corollary 26.** We can decide the ergodicity of a marked, live and open stochastic \( \Pi^3 \)-net in polynomial time.

### 4.2 Computing the steady-state distribution

In case the \( \Pi^3 \)-net is ergodic, it remains to compute its steady-state distribution, given by \( \pi(m) = \|v\|^{-1} v(m) \) for all \( m \in \mathcal{R}(m_0) \). Since we already computed \( \mathcal{R}(m_0) \) and \( v(m) \) for all markings \( m \in \mathcal{R}(m_0) \), it remains to compute the normalising constant \( \|v\| \).

This section is devoted to proving the following result.

**Theorem 27.** Let \((\mathcal{N}, \lambda, m_0)\) be a live ergodic stochastic \( \Pi^3 \)-net. There exists an algorithm for computing the normalising constant \( \|v\| \) in polynomial time with respect to \(|P|, |T|, W, \) and \( \|m_0\| \) (thus in pseudo-polynomial time).

This theorem applies to both closed and open \( \Pi^3 \)-nets. For closed nets, it provides a similar yet stronger result than Theorem 13, where the polynomial-time complexity was obtained only for a fixed value of \( N \) (the number of layers of the net).

We prove below Theorem 27 in the case of open nets. The case of closed nets is arguably easier: one can transform a closed net into an equivalent open net by adding a layer \( N + 1 \) with one place (and one virtual place \( p_{\text{eq}} \)), and set a firing rate \( \lambda_t = 0 \) for all transitions \( t \) of the layer \( N + 1 \). We therefore assume for the rest of this section that \( N \) is open.

We first describe a naive approach. The normalisation constant \( \|v\| \) can be computed as follows. Recall the family \( \mathcal{F} \) introduced in the proof of Theorem 24. We may prove that the set \( \mathcal{R}(m_0) \) is a union of (exponentially many) translated copies of the lattice \( \mathcal{L} \) generated by \( \mathcal{F} \). These copies may intersect each other, yet their intersections themselves are translated copies of \( \mathcal{L} \). Hence, using an inclusion-exclusion formula and a doubly exponential computation step, computing the sum \( \|v\| = \sum_{m \in \mathcal{R}(m_0)} \hat{v}(m) \) reduces to computing the sum \( \sum_{\ell \in \mathcal{L}} \hat{v}(\ell) \).

The family \( \mathcal{F} \) is not free a priori, hence computing this latter sum is not itself immediate. Using again inclusion-exclusion formulæ, we may write \( \mathcal{L} \) as a finite, disjoint union of exponentially many lattices generated by free subfamilies of \( \mathcal{F} \). This last step allows us to compute \( \sum_{\ell \in \mathcal{L}} \hat{v}(\ell) \), and therefore \( \|v\| \).

Such an approach suffers from a prohibitive computational cost. Yet it is conceptually simple, and it allows proving rather easily that \( \|v\| \) is a rational fraction in the constants \( \mu_p \), whose denominator is the product \( \prod_{\ell \in \mathcal{F}} (1 - \hat{v}(\ell)) \).
Example 28. Going back to the open $\Pi^3$-net of Figure 1, with initial marking $m_0 = q_3 + r_0$, this algorithm allows us to compute

$$
\|v\| = \frac{\mu_p a + (\mu_r a + \mu_r c) b}{c},
$$

with

$$
a = \mu_{p^2} \mu_{p^2} \mu_{q_1} \mu_{q_2} \mu_{q_3} + \mu_{p^2} \mu_{p^2} \mu_{q_1} \mu_{q_2} \mu_{q_3} + \mu_{q_1} \mu_{q_2} \mu_{q_3} + \mu_{q_1} \mu_{q_2} \mu_{q_3} + \mu_{q_1} \mu_{q_2} \mu_{q_3},
$$

$$
b = \mu_{p^2} \mu_{p^2} \mu_{q_1} \mu_{q_2} \mu_{q_3} + \mu_{p^2} \mu_{p^2} \mu_{q_1} \mu_{q_2} \mu_{q_3} + \mu_{p^2} \mu_{p^2} \mu_{q_1} \mu_{q_2} \mu_{q_3} + \mu_{p^2} \mu_{p^2} \mu_{q_1} \mu_{q_2} \mu_{q_3} + \mu_{p^2} \mu_{p^2} \mu_{q_1} \mu_{q_2} \mu_{q_3},
$$

and

$$
c = (1 - \mu_{p^2} \mu_{p^2} \mu_{p^2})(1 - \mu_{p^2} \mu_{p^2} \mu_{p^2})(1 - \mu_{q_1} \mu_{q_2} \mu_{p^2})(1 - \mu_{q_1} \mu_{q_2} \mu_{p^2}).
$$

Recall Example 25, which states that $\|v\|$ is finite if and only if all of $1 - \mu_{p^1}, 1 - \mu_{p^2} \mu_{p^2}, 1 - \mu_{q_2} \mu_{p^2}, 1 - \mu_{q_2} \mu_{p^2}$, and $1 - \mu_{p^2} \mu_{p^2}$ are positive. We observe that, as suggested above, the denominator $c$ is precisely the product of these five factors.

Our approach for computing $\|v\|$ will involve first computing variants of $\|v\|$. More precisely, for all subsets $Z$ of $P$, we consider a congruence relation $\sim_Z$ on markings, such that $m \sim_Z m'$ if and only if $m$ and $m'$ coincide on all places $p \in Z$. Then, we denote by $\mathfrak{M}_Z$ the quotient set $\mathbb{N}^P / \sim_Z$ and, for every element $m$ of $\mathfrak{M}_Z$, we denote by $\hat{\nu}(m)$ the product $\prod_{p \in Z} \mu_{P_{\nu}(p)}^{m(p)}$.

Two remarkable subsets $Z$ are the set $X = \bigcup_{i \in N - 1} P_i \cup P_{N - 1}^{\text{max}}$ and its complement $Y = P_{N - 1}^{\text{max}} \setminus P_N = P \setminus X$, which was already mentioned in Section 4.1. Indeed, places in $X$ are necessarily bounded while, if the net is unbounded, then so are the places in $Y$.

Based on these objects, and for all integers $c \geq 0$, we define two sets $\mathcal{C}_{m_0}(c)$ and $\mathcal{D}_{m_0}(c)$, which are respective subsets of $\mathfrak{M}_X$ and of $\mathfrak{M}_Y$. In some sense, the sets $\mathcal{C}_{m_0}(c)$ are meant to describe the “bounded” part of the markings in $\mathcal{R}(m_0)$, whose “unbounded” part is described by the sets $\mathcal{D}_{m_0}(c)$.

Definition 29. The set $\mathcal{C}_{m_0}(c)$ is the set of those classes $m_X$ in $\mathfrak{M}_X$ such that $m_X \cdot P_{N - 1}^{\text{max}} = c$, and such that $m_X$ contains some marking in $\mathcal{R}(m_0)$. The set $\mathcal{D}_{m_0}(c)$ is the set of those classes $m_Y$ in $\mathfrak{M}_Y$ such that $c + m_Y \cdot P_{N - 1}^{\text{max}} + m_Y \cdot \text{cin}(P_N) = c$, and such that $m_Y$ contains some marking in $\mathcal{R}(m_0)$.

These two sets of classes allow us to split nicely the huge sum $\|v\|$ into smaller independent sums, as stated in the result below. This decomposition result has the flavour of convolution algorithms, yet it requires a specific treatment since its terms are infinite sums.

Lemma 30. The normalisation constant $\|v\|$ is equal to the following finite sum:

$$
\|v\| = \sum_{c=0}^{\|P\|} \left( \sum_{m_X \in \mathcal{C}_{m_0}(c)} \hat{\nu}(m_X) \right) \left( \sum_{m_Y \in \mathcal{D}_{m_0}(c)} \hat{\nu}(m_Y) \right).
$$

It remains to compute sums $\sum_{m_X \in \mathcal{C}_{m_0}(c)} \hat{\nu}(m_X)$ and $\sum_{m_Y \in \mathcal{D}_{m_0}(c)} \hat{\nu}(m_Y)$ for polynomials many values of $c$. It turns out that such computations can be performed in polynomial time, as mentioned in Propositions 31 and 32.

Proposition 31. Let $(N, \lambda, m_0)$ be a live ergodic stochastic open $\Pi^3$-net, and let $c \geq 0$ be an integer. There exists an algorithm for computing the sum $\sum_{m_X \in \mathcal{C}_{m_0}(c)} \hat{\nu}(m_X)$ in polynomial time with respect to $|P|, |T|, W, c$ and $\|m_0\|$.

Proposition 31 is similar to the results of [10], and the algorithm can be adapted to closed $\Pi^3$-nets. The only major improvement here is that, instead of obtaining an algorithm that is polynomial-time for fixed values of $N$ only, our choice of invariants leads to a polynomial-time algorithm independently of $N$. 

CONCUR 2017
**Proposition 32.** Let \((N, \lambda, m_0)\) be a live ergodic stochastic open \(\Pi^1\)-net, and let \(c \geq 0\) be an integer. There exists an algorithm for computing the sum \(\sum_{m \in \mathcal{D}_{m_0}(c)} \hat{v}(m_Y)\) in polynomial time with respect to \(|P|, |T|, W, c\) and \(\|m_0\|\).

**Proof (sketch).** We compute the sum \(\sum_{m \in \mathcal{D}_{m_0}(c)} \hat{v}(m_Y)\) by using a dynamic-programming approach. Let \(a = \lfloor P_{\text{max}}/N - 1 \rfloor\), \(b = \lfloor |P_N| \rfloor\), and define \(t\) and \(u\) to be integers such that (i) \(\text{cin}(p_N^i) > 0\) iff \(1 \leq i \leq t\), (ii) \(\text{cin}(p_N^i) = 0\) iff \(t < i < u\) and (iii) \(\text{cin}(p_N^u) < 0\) iff \(u \leq i \leq b\). In addition, for all \(i \leq b\), let \(\Delta_i = \max\{1, |\text{cin}(p_N^i)|\}\). We consider below the lattice \(L = \mathbb{Z}^{u-t} \times \prod_{i=t+1}^u \Delta_i \mathbb{Z}\).

Now, consider integers \(A, B \in \mathbb{Z}\) and \(1 \leq \alpha \leq \alpha, 1 \leq \gamma \leq \beta \leq b\), as well as a vector \(w\) in the quotient set \(\mathbb{Z}^{u+b-1}/L\). We define auxiliary sets of vectors of the form

\[
\mathcal{D}(A, B, w, \alpha, \beta, \gamma) = \left\{ xy \in \mathbb{N}^{u+b-1} \bigg| \begin{array}{l}
A + \sum_{j=u}^\alpha y_j \geq i = 1 x_i + \sum_{j=\gamma}^\beta y_j \\
B + \sum_{j=u}^\beta y_j \geq \sum_{j=\gamma}^\gamma y_j \\
x_i = 0 \text{ for all } i > \alpha \\
y_j = 0 \text{ for all } j < \gamma \text{ and all } j > \beta
\end{array} \right\},
\]

where \(xy\) denotes the vector \((x_1, \ldots, x_{a-1}, y_1, \ldots, y_b)\).

First, we exhibit a bijection \(m \mapsto \overline{m}\), from the set \(\mathcal{D}_{m_0}(c)\) to a finite union of sets \(\mathcal{D}(A, B, 0, a-1, b, s)\), where the integers \(A, B\) and \(s\) can be computed efficiently. We can also prove a relation of the form \(\hat{v}(\overline{m}) = \hat{v}(\overline{\overline{m}})\) for all markings \(m \in \mathcal{D}_{m_0}(c)\), where \(\hat{v}\) is a product-form function \(\mathcal{T} : \mathbb{N}^{a+b-1} \rightarrow \mathbb{P}^{a-1}_x \mathbb{P}^b_y \mathbb{P}^{u-t}_y\) such that the constants \(P, \nu_{x,i}, \text{ and } \nu_{y,j}\) can be computed efficiently. It remains to compute sums of the form

\[U(A, B, w, \alpha, \beta, \gamma) = \sum_{xy \in \mathcal{D}(A, B, w, \alpha, \beta, \gamma)} \hat{v}(xy).\]

We do it by using recursive decompositions of the sets \(\mathcal{D}(A, B, w, \alpha, \beta, \gamma)\), where the integers \(A, B\) belong to the finite set \(\{-|A| - |B|, \ldots, |A| + |B|\}\), and the vector \(w\) is constrained to have at most one non-zero coordinate, chosen from some finite domain. These decompositions involve computing polynomially many auxiliary sums, which will prove Proposition 32. □

We illustrate the last part of this sketch of proof in the case where \(\gamma \leq t\), \(u \leq b\), and \(w\) is of the form \(\lambda 1_{y,\beta}\), for some \(\lambda \in \mathbb{Z} / \Delta_2 \mathbb{Z}\) (where we denote by \(1_{y,\beta}\) the vector of the canonical basis whose unique non-zero entry is \(y_j\)). In this case, we show how the sum \(U(A, B, w, \alpha, \beta, \gamma)\) can be expressed in terms of “smaller” sums \(U(A', B', w', \alpha', \beta', \gamma')\).

Let us split the set \(\mathcal{D}(A, B, w, \alpha, \beta, \gamma)\) into two subsets:

\[
\mathcal{D}_B(A, B, w, \alpha, \beta, \gamma) = \{ xy \in \mathcal{D}(A, B, w, \alpha, \beta, \gamma) \mid y_\beta \leq y_\gamma \}\]

and

\[
\mathcal{D}_C(A, B, w, \alpha, \beta, \gamma) = \{ xy \in \mathcal{D}(A, B, w, \alpha, \beta, \gamma) \mid y_\beta \geq y_\gamma \}, \text{ whose intersection is}
\]

\[
\mathcal{D}_D(A, B, w, \alpha, \beta, \gamma) = \{ xy \in \mathcal{D}(A, B, w, \alpha, \beta, \gamma) \mid y_\beta = y_\gamma \}.
\]

Computing \(U(A, B, w, \alpha, \beta, \gamma)\) amounts to computing the three sums \(\sum_{xy} \hat{v}(xy)\), where \(xy\) ranges respectively over \(\mathcal{D}_B(A, B, w, \alpha, \beta, \gamma)\), \(\mathcal{D}_C(A, B, w, \alpha, \beta, \gamma)\), and \(\mathcal{D}_D(A, B, w, \alpha, \beta, \gamma)\). We show here how to compute the first sum, which we simply denote by \(U_B\). The two latter sums are computed similarly.

Splitting every vector \(xy\) into one vector \(y_\beta(1_{y,\gamma} + 1_{y,\beta})\) and one vector \(xy' = xy - y_\beta(1_{y,\gamma} + 1_{y,\beta})\), we observe that \(xy - \lambda 1_{y,\beta} \in \mathbb{L}\) if and only if (i) \(y_\beta \equiv \lambda \mod \Delta_2\), and (ii)
xy' - yβ1y,γ ∈ L. It follows that
\[ \mathcal{D}_\oplus(A, B, \lambda 1y,\alpha, \beta, \gamma) = \bigcup_{j \geq 0} \{ xy \in \mathcal{D}_\oplus(A, B, \lambda 1y,\alpha, \beta, \gamma) \mid y_\beta = j \} \]
\[ = \bigcup_{j \geq 0, j \equiv \lambda \bmod \Delta_\beta} (\mathcal{D}(A, B, j 1y,\alpha, \beta - 1, \gamma) + j(1y,\alpha + y,\beta)). \]

Then, using the change of variable \( j = k + \Delta_\gamma \Delta_\beta \ell \) (with \( 0 \leq k < \Delta_\gamma \Delta_\beta \)), observe that \( j \equiv k \bmod \Delta_\beta \) and that \( j 1y,\alpha \equiv k 1y,\gamma \bmod L \), whence
\[ \mathcal{U}_\oplus = \sum_{j \geq 0} 1 \equiv \lambda \bmod \Delta_\beta \nu_{y,\alpha}^j \nu_{y,\beta}^j \mathcal{U}(A, B, j 1y,\alpha, \beta - 1, \gamma) \]
\[ = \sum_{\ell \geq 0} \sum_{k=0}^{\Delta_\gamma \Delta_\beta - 1} 1 \equiv \lambda \bmod \Delta_\beta (\nu_{y,\alpha} \nu_{y,\beta})^k \Delta_\gamma \Delta_\beta \mathcal{U}(A, B, k 1y,\alpha, \beta - 1, \gamma) \]
\[ = \frac{1}{1 - (\nu_{y,\alpha} \nu_{y,\beta})^{\Delta_\gamma \Delta_\beta}} \sum_{k=0}^{\Delta_\gamma \Delta_\beta - 1} 1 \equiv \lambda \bmod \Delta_\beta (\nu_{y,\alpha} \nu_{y,\beta})^k \mathcal{U}(A, B, k 1y,\alpha, \beta - 1, \gamma). \]

This shows, in this specific case, that computing \( \mathcal{U}_\oplus \) reduces to computing finitely many sums of the form \( \mathcal{U}(A, B, w,\alpha, \beta - 1, \gamma) \). Similar constructions are successfully used in all other cases.

## 5 Conclusion

Performance analysis of infinite-state stochastic systems is a very difficult task. Already checking the ergodicity is difficult in general, and even for systems which are known to be ergodic and which have product-form steady-state distributions, computing the normalising constant can be hard. In this work, we have proposed the model of open \( \Pi^3 \)-nets; this model generalises queuing networks and closed product-form Petri nets and generates a potentially infinite state-space. We have shown that we can efficiently decide (in polynomial time!) many behavioural properties, like the boundedness, the reachability in live nets, and the most important quantitative property: ergodicity. Furthermore, using dynamic programming algorithms managing infinite sums, we have shown that we can compute the normalising constant of the steady-state distribution in pseudo-polynomial time.

We believe our approach can be extended to \( \Pi^3 \)-nets in which one place is removed in every layer, without affecting too much the complexity. This setting would allow to model production of resources by the environment while in the current version resources may grow in an unbounded way but only when the number of processes of the main layer also grows. We leave this as future work.

## References

Unbounded Product-Form Petri Nets


Efficient Coalgebraic Partition Refinement∗†

Ulrich Dorsch1, Stefan Milius2, Lutz Schröder3, and Thorsten Wißmann4

1 Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany
   Ulrich.Dorsch@fau.de
2 Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany
   mail@stefan-milius.eu, http://orcid.org/0000-0002-2021-1644
3 Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany
   Lutz.Schroeder@fau.de
4 Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany
   Thorsten.Wissmann@fau.de

Abstract
We present a generic partition refinement algorithm that quotients coalgebraic systems by
behavioural equivalence, an important task in reactive verification; coalgebraic generality implies in
particular that we cover not only classical relational systems but also various forms of weighted
systems. Under assumptions on the type functor that allow representing its finite coalgebras in
terms of nodes and edges, our algorithm runs in time \( O(m \cdot \log n) \) where \( n \) and \( m \) are the numbers
of nodes and edges, respectively. Instances of our generic algorithm thus match the runtime of
the best known algorithms for unlabelled transition systems, Markov chains, and deterministic
automata (with fixed alphabets), and improve the best known algorithms for Segala systems.

1998 ACM Subject Classification F.1.1 Models of Computation, F.1.2 Modes of Computation

Keywords and phrases coalgebra, markov chains, partition refinement, transition systems

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2017.32

1 Introduction

Minimization under bisimilarity is the task of identifying all states in a reactive system that
exhibit the same behaviour. Minimization appears as a subtask in state space reduction
(e.g. [5]) or non-interference checking [34]. The notion of bisimulation was first defined
for relational systems [33, 24, 26]; it was later extended to other system types including
probabilistic systems [23, 9] and weighted automata [6]. In fact, the importance of minimiza-
tion under bisimilarity appears to increase with the complexity of the underlying system
type. E.g., while in LTL model checking, minimization drastically reduces the state space
but, depending on the application, does not necessarily lead to a speedup in the overall
balance [11], in probabilistic model checking, minimization under strong bisimilarity does
lead to substantial efficiency gains [19].

The algorithmics of minimization, often referred to as partition refinement or lumping,
has received a fair amount of attention. Since bisimilarity is a greatest fixpoint, it is more or
less immediate that it can be calculated in polynomial time by approximating this fixpoint
from above following Kleene’s fixpoint theorem. In the relational setting, Kanellakis and
Smolka [18] introduced an algorithm that in fact runs in time $O(nm)$ where $n$ is the number of nodes and $m$ is the number of transitions. An even more efficient algorithm running in time $O(m \log n)$ was later described by Paige and Tarjan [25]; this bound holds even if the number of action labels is not fixed [31]. Current algorithms typically apply further optimizations to the Paige-Tarjan algorithm, thus achieving better average-case behaviour but the same worst-case behaviour [10]. Probabilistic minimization has undergone a similarly dynamic development [3, 7, 36], and the best algorithms for minimization of Markov chains now have the same $O(m \log n)$ run time as the relational Paige-Tarjan algorithm [15, 8, 32]. Using ideas from abstract interpretation, Ranzato and Tapparo [27] have developed a relational partition refinement algorithm that is generic over notions of process equivalence. As instances, they recover the classical Paige-Tarjan algorithm for strong bisimilarity and an algorithm for stuttering equivalence, and obtain new algorithms for simulation equivalence and for a new process equivalence.

In this paper we follow an orthogonal approach and provide a generic partition refinement algorithm that can be instantiated for many different types of systems (e.g. nondeterministic, probabilistic, weighted). We achieve this by methods of universal coalgebra [28]. That is, we encapsulate transition types of systems as endofunctors on sets (or a more general category), and model systems as coalgebras for a given type functor.

Our work proceeds on several levels of abstraction. On the most abstract level (Section 3) we work with coalgebras for a monomorphism-preserving endofunctor on a category with image factorizations. Here we present a quite general category-theoretic partition refinement algorithm, and we prove its correctness. The algorithm is parametrized over a select routine that determines which observations are used to split blocks of states; the corner case where all available observations are used yields known coalgebraic final chain algorithms, e.g. [22].

Next, we present an optimized version of our algorithm (Section 4) that needs more restrictive conditions to ensure correctness; specifically, we need to assume that the type endofunctor satisfies a condition we call zippability in order to allow for incremental computation of partitions. This property holds, e.g., for all polynomial endofunctors on sets and for the type functors of labelled and weighted transition systems, but not for all endofunctors of interest. In particular, zippable functors fail to be closed under composition, as exemplified by the double covariant powerset functor $\mathbb{P}\mathbb{P}$ on sets, for which the optimized algorithm is in fact incorrect. However, it turns out that obstacles of this type can be removed by moving to multi-sorted coalgebras [29], so we do eventually obtain an efficient partition refinement algorithm for coalgebras of composite functors, including $\mathbb{P}\mathbb{P}$-coalgebras as well as (probabilistic) Segala systems [30].

Finally, we analyse the run time of our algorithm (Section 5). To this end, we make our algorithm parametric in an abstract refinement interface to the type functor, which encapsulates the incremental calculation of partitions in the optimized version of the algorithm. We show that if the interface operations can be implemented in linear time, then the algorithm runs in time $O(m \log n)$, where $n$ is the number of states and $m$ the number of `edges’ in a syntactic encoding of the input coalgebra. We thus recover the most efficient known algorithms for transition systems (Paige and Tarjan [25]) and for weighted systems (Valmari and Franceschinis [32]). Using the mentioned modularity results, we also obtain an $O((m + n) \log (m + n))$ algorithm for Segala systems, to our knowledge a new result (more precisely, we improve an earlier bound established by Baier, Engelen, and Majster-Cederbaum [3], roughly speaking by letting only non-zero probabilistic edges enter into the time bound). The algorithm and its analysis apply also to generalized polynomial functors on sets; in particular, for the functor $2 \times (\cdot)^4$, which models deterministic finite automata, we obtain the same complexity $O(n \log n)$ as for Hopcroft’s classical minimization algorithm for a fixed alphabet $A$ [14, 21, 12].
2 Preliminaries

We assume that readers are familiar with basic category theory [2]. For the convenience of the reader we recall some concepts that are central for the categorical version of the algorithm.

► Notation 2.1. The terminal object is denoted by $1$, with unique arrows $!: A \rightarrow 1$, and the product of objects $A$, $B$ by $A \times B \xrightarrow{\pi_1} A \times B \xrightarrow{\pi_2} B$. Given $f : D \rightarrow A$ and $g : D \rightarrow B$, the morphism induced by the universal property of the product $A \times B$ is denoted by $(f, g) : D \rightarrow A \times B$. The kernel $\ker f$ of a morphism $f$ is the pullback of $f$ along itself. We write $\twoheadrightarrow$ for regular epimorphisms (i.e. coequalizers), and $\mapsto$ for monomorphisms.

Kernels allow us to talk about equivalence relations in a category. In particular in $\text{Set}$, there is a bijection between kernels and equivalence relations in the usual sense: For a map $f : D \rightarrow A$, $\ker f = \{(x, y) \mid fx = fy\}$ is the equivalence relation induced by $f$. Generally, relations (i.e. jointly monic spans of morphisms) in a category are ordered by inclusion in the obvious way. We say that a kernel $K$ is finer than a kernel $K'$ if $K$ is included in $K'$.

We use intersection $\cap$ and union $\cup$ of kernels for meets and in joins in the inclusion ordering on relations (not equivalence relations or kernels); in this notation, $\ker (f, g) = \ker f \cap \ker g$. In $\text{Set}$, a map $f : D \rightarrow A$ factors through the partition $D/\ker f$ induced by its kernel, via the map $[-] : D \rightarrow D/\ker f$ taking equivalence classes

$$[x]_f := \{x' \in D \mid fx = fx'\} = \{x' \in D \mid (x, x') \in \ker f\}.$$ 

Well-definedness of functions on $D/\ker f$ is determined precisely by the universal property of $[-]_f$ as a coequalizer of $\ker f \twoheadrightarrow D$. In particular, $f$ induces an injection $D/\ker f \hookrightarrow A$; together with $[-]_f$, this is the factorization of $f$ into a regular epimorphism and a monomorphism. Categorically, this is captured by the following assumptions.

► Assumption 2.2. We assume throughout that $C$ is a finitely complete category that has coequalizers and image factorizations, i.e. every morphism $f$ has a factorization $f = m \cdot e$ as a regular epimorphism $e$ followed by a monomorphism $m$. We call the codomain of $e$ the image of $f$, and denote it by $D/\ker f$. Regular epis in $C$ are closed under composition and right cancellation [2, Prop. 14.14].

► Examples 2.3. Examples of categories satisfying Assumption 2.2 abound. In particular, every regular category with coequalizers satisfies our assumptions. The category $\text{Set}$ of sets and maps is, of course, regular. Every topos is regular, and so is every finitary variety, i.e. a category of algebras for a finitary signature satisfying given equational axioms (e.g. monoids, groups, vector spaces etc.). Posets and topological spaces fail to be regular but still satisfy our assumptions. If $C$ is regular, so is the functor category $C^E$ for any category $E$.

For a set $S$ of sorts, the category $\text{Set}^S$ of $S$-sorted sets has $S$-tuples of sets as objects. We write $\chi_S : X \rightarrow 2$ for the characteristic function of a subset $S \subseteq X$, i.e. for $x \in X$ we have $\chi_S(x) = 1$ if $x \in S$ and $\chi_S(x) = 0$ otherwise. We will also use a three-valued version:

► Definition 2.4. For $S \subseteq C \subseteq X$, define $\chi^C_S : X \rightarrow 3$ by $C \ni x \mapsto 0$, $C \setminus S \ni x \mapsto 1$, and $S \ni x \mapsto 2$. (This is essentially $(\chi_S, \chi_C) : X \rightarrow 4$ without the impossible case $x \in S \setminus C$.)

Coalgebras. We briefly recall basic notions from coalgebra. For introductory texts, see [28, 17, 1, 16]. Given an endofunctor $H : C \rightarrow C$, a coalgebra is a pair $(C, c)$ where $C$ is an object of $C$ called the carrier and thought of as an object of states, and $c : C \rightarrow HC$ is a morphism called the structure of the coalgebra. Our leading examples are the following.
Example 2.5.

1. Labelled transition systems with labels from a set $A$ are coalgebras for the functor $HX = \mathcal{P}(A \times X)$ (and unlabelled transition systems are simply coalgebras for $\mathcal{P}$). Explicitly, a coalgebra $c : C \to HC$ assigns to each state $x$ a set $c(x) \in \mathcal{P}(A \times X)$, and this represents the transition structure at $x$: $x$ has an $a$-transition to $y$ iff $(a, y) \in c(x)$.

2. Weighted transition systems with weights drawn from a commutative monoid are modelled as coalgebras as follows. For the given commutative monoid $(M, +, 0)$, we consider the monoid-valued functor $M(-)$ on $\text{Set}$ given for any map $h : X \to Y$ by

$$M^{(X)} = \{ f : X \to M \mid f(x) \neq 0 \text{ for finitely many } x \}, \quad M^{(H)}(f)(y) = \sum_{h(x)=y} f(x).$$

$M$-weighted transition systems are in bijective correspondence with coalgebras for $M(-)$ [13] (and for $M$-weighted labelled transition systems one takes $(M(-))^A$).

3. Probabilistic transition systems are modelled coalgebraically using the distribution functor $\mathcal{D}$. This is the subfunctor $\mathcal{D}X \subseteq \mathbb{R}_\geq^{(X)}$, where $\mathbb{R}_\geq$ is the monoid of addition on the non-negative reals, given by $\mathcal{D}X = \{ f \in \mathbb{R}_\geq^{(X)} \mid \sum_{x \in X} f(x) = 1 \}$.

4. The finite powerset functor $\mathcal{P}_f$ is a monoid-valued functor for the Boolean monoid $\mathbb{B} = (2, v, 0)$. The bag functor $\mathcal{B}_c$, which assigns to a set $X$ the set of bags (i.e. finite multisets) on $X$, is the monoid-valued functor for the additive monoid of natural numbers.

5. Simple (resp. general) Segala systems [30] strictly alternate between non-deterministic and probabilistic transitions; they can be modeled as coalgebras for the set functor $\mathcal{P}(\times D(\cdot))$ (resp. $\mathcal{P}_f(\times D(\cdot))$).

A coalgebra morphism from a coalgebra $(C, c)$ to a coalgebra $(D, d)$ is a morphism $h : C \to D$ such that $d \cdot h = Hh \cdot c$; intuitively, coalgebra morphisms preserve observable behaviour. Coalgebras and their morphisms form a category $\text{Coalg}(H)$. The forgetful functor $\text{Coalg}(H) \to \mathcal{C}$ creates all colimits, so $\text{Coalg}(H)$ has all colimits that $\mathcal{C}$ has.

A subcoalgebra of a coalgebra $(C, c)$ is represented by a coalgebra morphism $m : (C, c) \to (D, d)$ such that $m$ is a monomorphism in $\mathcal{C}$. Likewise, a quotient of a coalgebra $(C, c)$ is represented by a coalgebra morphism $q : (C, c) \to (D, d)$ carried by a regular epimorphism $q$ of $\mathcal{C}$. If $H$ preserves monomorphisms, then the image factorization structure on $\mathcal{C}$ lifts to coalgebras.

Definition 2.6. A coalgebra is simple if it does not have any non-trivial quotients.

Equivalently, a coalgebra $(C, c)$ is simple if every coalgebra morphism with domain $(C, c)$ is carried by a monomorphism. Intuitively, in a simple coalgebra all states exhibiting the same observable behaviour are already identified. This paper is concerned with the design of algorithms for computing the simple quotient of a given coalgebra:

Lemma 2.7. The simple quotient of a coalgebra is unique (up to isomorphism).

Intuitively speaking, two elements (possibly in different coalgebras) are called behaviourally equivalent if they can be identified by coalgebra morphisms. Hence, the simple quotient of a coalgebra is its quotient modulo behavioural equivalence. In our main examples, this means that we minimize w.r.t. standard bisimilarity-type equivalences.

Example 2.8. Behavioural equivalence instantiates to various notions of bisimilarity:

1. Park-Milner bisimilarity on labelled transition systems;
2. weighted bisimilarity on weighted transition systems [20, Proposition 2];
3. stochastic bisimilarity on probabilistic transition systems [20];
4. Segala bisimilarity on simple and general Segala systems [4, Theorem 4.2].
3 A Categorical Algorithm for Behavioural Equivalence

We proceed to describe a categorical partition refinement algorithm that computes the simple quotient of a given coalgebra under fairly general assumptions.

Assumption 3.1. Assume that $H$ is an endofunctor on $C$ that preserves monomorphisms.

Note that mono preservation is w.l.o.g. for $C = \text{Set}$. Roughly, for a given coalgebra $\xi : X \to HX$ in $\text{Set}$, a partition refinement algorithm maintains a quotient $q : X \to X/Q$ that distinguishes some (but possibly not all) states with different behaviour, and in fact, initially $q$ typically identifies everything. The algorithm repeats the following steps:

1. Gather new information on which states should become separated by using $X \xrightarrow{\xi} HX \xrightarrow{Hz} HX/Q$, i.e., by identifying equivalence classes under $q$ that contain states whose behaviour is observed to differ under one more step of the transition structure $\xi$.

2. Use parts of this information to refine $q$ and repeat until $q$ does not change any more.

One of the core ideas of the Paige-Tarjan partition refinement algorithm [25] is to not use all information immediately in the second step. Recall that the algorithm maintains two partitions $Y$ and $Z$ of the state set $X$ of the given transition system; the elements of $Y$ are called subblocks and the elements of $Z$ are called compound blocks. The partition $Y$ is a refinement of the partition $Z$. The key to the time efficiency of the algorithm is to select in each iteration a subblock that is at most half of the size of the compound block it belongs to. At the present high level of generality (which in particular does not know about sizes of objects), we encapsulate the subblock selection in a routine select, assumed as a parameter to our algorithm:

Definition 3.2. A select routine is an operation that receives a chain of two regular epis $X \xrightarrow{\chi_S} Y \xrightarrow{\chi_{\xi(S)}} Z$ and returns some morphism $k : Y \to K$ into some object $K$. We call $Y$ the subblocks and $Z$ the compound blocks.

The idea is that the morphism $k$ throws away some of the information provided by the refinement $Y$. For example, in the Paige-Tarjan algorithm it models the selection of one compound block to be split in two parts, which then induce the further refinement of $Y$.

Example 3.3.

1. In the classical Paige-Tarjan algorithm [25], i.e., for $C = \text{Set}$, one wants to find a proper subblock that is at most half of the size of the compound block it sits in. So let $S \in Y$ such that $2 \cdot |y^{-1}([S])]| \leq |(zy)^{-1}([z(S)]]|$. Here, $z(S)$ is the compound block containing $S$. Then we let select$(x, y)$ be $k : Y \to 3$ given by $k(x) = 2$ if $x = S$, else $k(x) = 1$ if $z(x) = z(S)$, and $k(x) = 0$ otherwise; i.e. $k = \chi_S^C$ (Theorem 2.4). If $Y$ and $Z$ are encoded as partitions of $X$, then $S$ and $C := z(S)$ are subsets of $X$ and $k \cdot y = \chi_S^C$. If there is no such $S \in Y$, then $z$ is bijective, i.e., there is no compound block from $Z$ that needs to be refined. In this case, $k$ does not matter and we simply put $k = ! : Y \to 1$.

2. One obvious choice for $k$ is to take the identity on $Y$, so that all of the information present in $Y$ is used for further refinement. We will discuss this in Theorem 3.

3. Two other, trivial, choices are $k = ! : Y \to 1$ and $k = z$. Since both of these choices provide no extra information, this will leave the partitions unchanged, see Theorem 3.12.

Algorithm 3.4. Given a coalgebra $\xi : X \to HX$, we successively refine equivalence relations $Q$ and $P$ on $X$, maintaining the invariant that $P$ is finer than $Q$. In each step, we take into account new information on the behaviour of states, represented by a map
q : X → K, and accumulate this information in a map \( \bar{q} : X \to \bar{K} \). To facilitate the analysis, these variables are indexed over loop iterations in the description. Initial values are

\[
Q_0 = X \times X \quad q_0 = ! : X \to 1 = K_0 \quad P_0 = \ker(X \xrightarrow{\xi} HX \xrightarrow{H} H1).
\]

We then iterate the following steps while \( P_i \neq Q_i \), for \( i \geq 0 \):

1. \( X/P_i \xrightarrow{k_{i+1}} K_{i+1} := \text{select}(X \to X/P_i \to X/Q_i) \), using that \( X/P_i \) is finer than \( X/Q_i \)
2. \( q_{i+1} := X \xrightarrow{\bar{q}_{i+1}} K_{i+1} \), \( \bar{q}_{i+1} := (\bar{q}_i, q_{i+1}) : X \to \bar{K}_i \times K_{i+1} \)
3. \( Q_{i+1} := \ker \bar{q}_{i+1} \) (\( = \ker(\bar{q}_i, q_{i+1}) = \ker \bar{q}_i \cap \ker q_{i+1} \))
4. \( P_{i+1} := \ker\big(X \xrightarrow{\xi} HX \xrightarrow{H\bar{q}_{i+1}} H \prod_{j \leq i+1} K_j\big)\)

Upon termination, the algorithm returns \( X/P_i = X/Q_i \) as the simple quotient of \( (X, \xi) \).

**Notation 3.5.** For spans \( R \rightrightarrows X \), we will denote the canonical quotient by \( \kappa_R \colon X \to X/R \).

We proceed to prove correctness, i.e. that the algorithm really does return the simple quotient of \((X, \xi)\). We fix the notation in Algorithm 3.4 throughout. Since \( \bar{q} \) accumulates more information in every step, it is clear that \( P \) and \( Q \) are really being successively refined:

**Lemma 3.6.** For every \( i \), \( P_{i+1} \) is finer than \( P_i \), \( Q_{i+1} \) is finer than \( Q_i \), and \( P_i \) is finer than \( Q_{i+1} \).

\[
\begin{array}{cccccccc}
Q_0 & \leftrightarrow & Q_1 & \leftrightarrow & Q_2 & \cdots & \leftrightarrow & Q_{i+1} & \leftrightarrow & Q_{i+2} & \leftrightarrow & \cdots \\
\uparrow & & \uparrow & & \uparrow & & \uparrow & & \uparrow & & \uparrow & \\
P_0 & \leftrightarrow & P_1 & \leftrightarrow & P_2 & \leftrightarrow & P_3 & \leftrightarrow & P_i & \leftrightarrow & P_{i+1} & \leftrightarrow & \cdots \\
\end{array}
\] (3.1)

If we suppress the termination on \( P_i = Q_i \), for a moment, then the algorithm thus computes equivalence relations refining each other. At each step, \text{select} decides which part of the information present in \( P_i \) but not in \( Q_i \) should be used to refine \( Q_i \) to \( Q_{i+1} \).

**Proposition 3.7.** There exist morphisms \( \xi/Q_i \colon X/P_i \to H(X/Q_i) \) for \( i \geq 0 \) (necessarily unique) such that (3.2) commutes.

\[
\begin{array}{ccc}
X & \xrightarrow{\kappa_{P_i}} & X/P_i \\
\downarrow^{\xi} & & \downarrow^{\xi/Q_i} \\
HX & \xrightarrow{H\kappa_{Q_i}} & H(X/Q_i)
\end{array}
\] (3.2)

Upon termination the morphism \( \xi/Q_i \) yields the structure of a quotient coalgebra of \( \xi \):

**Corollary 3.8.** If \( P_i = Q_i \) then \( X/Q_i \) carries a unique coalgebra structure forming a quotient of \( \xi : X \to HX \).

This means intuitively that all states that are merged by the algorithm are actually behaviourally equivalent. The following property captures the converse:

**Lemma 3.9.** Let \( h : (X, \xi) \to (D, d) \) be a quotient of \((X, \xi)\). Then \( \ker h \) is finer than both \( P_i \) and \( Q_i \), for all \( i \geq 0 \).

**Theorem 3.10** (Correctness). If \( P_i = Q_i \), then \( \xi/Q_i \colon X/Q_i \to H X/Q_i \) is a simple coalgebra.

**Remark.** Most classical partition refinement algorithms are parametrized by an initial partition \( \kappa_I : X \to X/I \). We start with the trivial partition \( ! : X \to 1 \) because a non-trivial initial partition might split equivalent behaviours and then would invalidate Theorem 3.9. To accomodate an initial partition \( X/I \) coalgebraically, replace \((X, \xi)\) with the coalgebra \( \langle \xi, \kappa_I \rangle \) for the functor \( H(-) \times X/I \) – indeed, already \( P_0 \) will then be finer than \( I \).
We look in more detail at two corner cases of the algorithm, where the select routine retains all available information, respectively none:

- **Remark.** Recall that \( H \) induces the final sequence:

\[
1 \leftarrow H1 \overset{H}{\leftarrow} H^2 \overset{H}{\leftarrow} \cdots \overset{H}{\leftarrow} H^i \overset{H}{\leftarrow} H^{i+1} \overset{H^i}{\leftarrow} \cdots
\]

Every coalgebra \( \xi : X \to HX \) then induces a canonical cone \( \xi^{(i)} : X \to H^i1 \) on the final sequence, defined inductively by \( \xi^{(0)}(x) = x \), \( \xi^{(i+1)} = H\xi^{(i)} \cdot \xi \). The objects \( H^n1 \) may be thought of as domains of \( n \)-step behaviour for \( H \)-coalgebras. If \( C = \text{Set} \) and \( X \) is finite, then states \( x \) and \( y \) are behaviourally equivalent if \( \xi^{(i)}(x) = \xi^{(i)}(y) \) for all \( i \leq \omega \) [35].

The vertical inclusions in (3.1) reflect that only some and not necessarily all of the information present in the relation \( P_i \) (resp. the quotient \( X/P_i \)) is used for further refinement. If indeed everything is used, i.e., we have \( k_{i+1} := \text{id}_{X/P_i} \), then these inclusions become isomorphisms and then our algorithm simply computes the kernels of the morphisms in the canonical cone, i.e. \( Q_i = \ker \xi^{(i)} \).

That is, when select retains all available information, then Algorithm 3.4 just becomes a standard final chain algorithm (e.g. [22]). The other extreme is the following:

- **Definition 3.11.** We say that select discards all new information at \( i + 1 \) if \( k_{i+1} \) factors through the morphism \( X/P_i \to X/Q_i \), witnessing that \( P_i \) is finer than \( Q_i \), see Theorem 3.6.

- **Lemma 3.12.** The algorithm fails to progress in the \( i + 1 \)-th iteration, i.e. \( Q_{i+1} = Q_i \), iff select discards all new information at \( i + 1 \).

- **Corollary 3.13.** If \( C \) is (concrete over) \( \text{Set}^S \) and select never discards all new information, then Algorithm 3.4 terminates and computes the simple quotient of a given finite coalgebra.

Indeed, Proposition 3.7 shows that we obtain a chain of successively finer quotients of \( X \), and by Theorem 3.12 this chain must finally converge (i.e. \( P_i = Q_i \) will hold).

### 4 Incremental Partition Refinement

In the most generic version of the partition refinement algorithm (Algorithm 3.4), the partitions are recomputed from scratch in every step: In Step 4 of the algorithm, \( P_{i+1} = \ker(H(\bar{q}_{i}, q_{i+1}) \cdot \xi) \) is computed from the information \( \bar{q}_{i} \) accumulated so far and the new information \( q_{i+1} \), but in general one cannot exploit that the kernel of \( \bar{q}_{i} \) has already been computed. We now present a refinement of the algorithm in which the partitions are computed incrementally, i.e. \( P_{i+1} \) is computed from \( P_i \) and \( q_{i+1} \). This requires the type functor \( H \) to be zippable (Theorem 4.1). The algorithm will be further refined in the next section.

Note that in Step 3, Algorithm 3.4 computes a kernel \( Q_{i+1} = \ker \bar{q}_{i+1} = \ker(\bar{q}_{i}, q_{i+1}) \). In general, the kernel of a pair \( \langle a, b \rangle : D \to A \times B \) is an intersection \( \ker a \cap \ker b \). Hence, the partition for such a kernel can be computed in two steps:

1. Compute \( D/\ker a \).
2. Refine every block in \( D/\ker a \) with respect to \( b : D \to B \).

Algorithm 3.4 can thus be implemented to keep track of the partition \( X/Q_i \) and then refine this partition by \( q_{i+1} \) in each iteration.

However, the same trick cannot be applied immediately to the computation of \( X/P_i \), because of the functor \( H \) inside the computation of the kernel: \( P_{i+1} = \ker(H(\bar{q}_{i}, q_{i+1}) \cdot \xi) \).

In the following, we will provide sufficient conditions for \( H, a : D \to A, b : D \to B \) to satisfy

\[
\ker H(a, b) = \ker(Ha, Hb).
\]
unzip

\[ \begin{align*}
(a_1, a_2, b_1, b_2) & \mapsto \{a_1, a_2, b_1\} \mapsto \{\{a_1, b_1\}, \{a_2, b_2\}\} \\
(a_1, a_2, a_3, \ldots, b_1, b_2) & \mapsto \{\{a_1, a_2, \ldots, \}, \{\ldots, b_1\}\} \mapsto \{\{\ldots, b_1\}, \{\ldots, b_2\}\} \\
\end{align*} \]

(a) \((-)^*\) is zippable. (b) \(P_i\) is zippable. (c) \(P_iP_t\) is not zippable.

\[\text{Figure 1} \quad \text{Zippability of Set-Functors for sets } A = \{a_1, a_2, a_3\}, B = \{b_1, b_2\}.\]

As soon as this holds for \(a = \bar{q}_1, b = q_{i+1}\), we can optimize the algorithm by changing Step 4 to

\[ P'_{i+1} := \ker(H\bar{q}_i \cdot \xi, Hq_{i+1} \cdot \xi). \]  

(4.1)

**Definition 4.1.** A functor \(H\) is zippable if the following morphism is a monomorphism:

\[ \text{unzip}_{H,A,B} : H(A + B) \xrightarrow{(H(A+1).H([+B]))} H(A + 1) \times H(1 + B) \]

Intuitively, if \(H\) is a functor on \(\text{Set}\), we think of elements \(t\) of \(H(A + B)\) as shallow terms with variables from \(A + B\). Then zippability means that each \(t\) is uniquely determined by the two terms obtained by replacing \(A\) and \(B\)-variables, respectively, by some placeholder \(\_\), viz. the element of 1, as in the examples in Figure 1.

In the following, we work in the category \(\mathcal{C} = \text{Set}^S\) of \(S\)-sorted sets. However, most proofs are category-theoretic to clarify where sets are really needed and where the arguments are more generic.

**Example 4.2.**

1. Constant functors \(X \mapsto A\) are zippable: unzip is the diagonal \(A \to A \times A\).
2. The identity functor is zippable since \(A + B \xrightarrow{(A+1) + (1+B)} (A + 1) \times (1 + B)\) is monic in \(\text{Set}^S\).
3. From Lemma 4.3 it follows that every polynomial endofunctor is zippable.

**Lemma 4.3.** Zippable endofunctors are closed under products, coproducts and subfunctors.

**Lemma 4.4.** If \(H\) has a componentwise monic natural transformation \(H(X + Y) \to HX \times HY\), then \(H\) is zippable.

**Example 4.5.**

1. For every commutative monoid, the monoid-valued functor \(M(-)\) admits a natural isomorphism \(M(X+Y) \cong M(X) \times M(Y)\), and hence is zippable by Lemma 4.4.
2. As special cases of monoid-valued functors we obtain that the finite powerset functor \(P_t\) and the bag functor \(B_t\) are zippable.
3. The distribution functor \(D\) (see Example 2.5) is a subfunctor of the monoid-valued functor \(M(-)\) for \(M\) the additive monoid of real numbers, and hence is zippable by Item 1 and Lemma 4.3.
4. The previous examples together with the closure properties in Lemma 4.3 show that a number of functors of interest are zippable, e.g. \(2 \times (-)^A\), \(2 \times P(-)^A\), \(P(A \times (-))\), \(2 \times ((-)+1)^A\), and variants where \(P\) is replaced with \(B_t, M(-)^{\ast}\), or \(D\).

**Example 4.6.** The finitary functor \(P_tP_t\) fails to be zippable, as shown in Figure 1. First, this shows that zippable functors are not closed under quotients, since any finitary functor is a quotient of a polynomial, hence zippable, functor (recall that a Set-functor \(F\) is finitary if \(FX = \bigcup\{F[Y] \mid i : Y \to X \text{ and } Y \text{ finite}\}\)). Secondly, this shows that zippable functors are not closed under composition. One can extend the counterexample to a coalgebra to show that the optimization is incorrect for \(P_tP_t\) and select = \(\chi_S^C\). We will remedy this later by making use of a second sort, i.e. by working in \(\text{Set}^S\) (Theorem 4).
with real-valued weights \[32\], it may in general let relations are closed under unions, the definition boils down to Definition 4.8. ▶

So if we computed \( P_2 \) iteratively as in (4.1) for \( q_2 = \chi_S \), then \( t_1 \) and \( t_2 \) would not be split, and we would reach the termination condition \( P_2 = Q_2 \) before all behaviourally inequivalent states have been separated.

Already Paige and Tarjan [25, Step 6 of the Algorithm] note that one additionally needs to split by \( C \setminus S = \{ c_3 \} \), which is accomplished by splitting by \( q_3 = \chi^C_S \). This is formally captured by the condition we introduce next.

\[ Hq_1 \cdot \xi(t_1) = \big( \text{id} \times \mathcal{P}\xi(Hq_0) \big) \cdot \xi(t_1) = (\big( \mathcal{P} \cdot \{ c_1, c_2 \} \big)) = Hq_1 \cdot \xi(t_2) \]

\[ Hq_2 \cdot \xi(t_1) = \big( \text{id} \times \mathcal{P}\chi_S(c_3) \big) \cdot \xi(t_1) = (\big( \{ 0,1 \} \big)) = Hq_2 \cdot \xi(t_2) \]

So if we computed \( P_2 \) iteratively as in (4.1) for \( q_2 = \chi_S \), then \( t_1 \) and \( t_2 \) would not be split, and we would reach the termination condition \( P_2 = Q_2 \) before all behaviourally inequivalent states have been separated.

Already Paige and Tarjan [25, Step 6 of the Algorithm] note that one additionally needs to split by \( C \setminus S = \{ c_3 \} \), which is accomplished by splitting by \( q_3 = \chi^C_S \). This is formally captured by the condition we introduce next.

\[ Hq_1 \cdot \xi(t_1) = \big( \text{id} \times \mathcal{P}\xi(Hq_0) \big) \cdot \xi(t_1) = (\big( \mathcal{P} \cdot \{ c_1, c_2 \} \big)) = Hq_1 \cdot \xi(t_2) \]

\[ Hq_2 \cdot \xi(t_1) = \big( \text{id} \times \mathcal{P}\chi_S(c_3) \big) \cdot \xi(t_1) = (\big( \{ 0,1 \} \big)) = Hq_2 \cdot \xi(t_2) \]

So if we computed \( P_2 \) iteratively as in (4.1) for \( q_2 = \chi_S \), then \( t_1 \) and \( t_2 \) would not be split, and we would reach the termination condition \( P_2 = Q_2 \) before all behaviourally inequivalent states have been separated.

Already Paige and Tarjan [25, Step 6 of the Algorithm] note that one additionally needs to split by \( C \setminus S = \{ c_3 \} \), which is accomplished by splitting by \( q_3 = \chi^C_S \). This is formally captured by the condition we introduce next.

\[ Hq_1 \cdot \xi(t_1) = \big( \text{id} \times \mathcal{P}\xi(Hq_0) \big) \cdot \xi(t_1) = (\big( \mathcal{P} \cdot \{ c_1, c_2 \} \big)) = Hq_1 \cdot \xi(t_2) \]

\[ Hq_2 \cdot \xi(t_1) = \big( \text{id} \times \mathcal{P}\chi_S(c_3) \big) \cdot \xi(t_1) = (\big( \{ 0,1 \} \big)) = Hq_2 \cdot \xi(t_2) \]

So if we computed \( P_2 \) iteratively as in (4.1) for \( q_2 = \chi_S \), then \( t_1 \) and \( t_2 \) would not be split, and we would reach the termination condition \( P_2 = Q_2 \) before all behaviourally inequivalent states have been separated.

Already Paige and Tarjan [25, Step 6 of the Algorithm] note that one additionally needs to split by \( C \setminus S = \{ c_3 \} \), which is accomplished by splitting by \( q_3 = \chi^C_S \). This is formally captured by the condition we introduce next.

\[ Hq_1 \cdot \xi(t_1) = \big( \text{id} \times \mathcal{P}\xi(Hq_0) \big) \cdot \xi(t_1) = (\big( \mathcal{P} \cdot \{ c_1, c_2 \} \big)) = Hq_1 \cdot \xi(t_2) \]

\[ Hq_2 \cdot \xi(t_1) = \big( \text{id} \times \mathcal{P}\chi_S(c_3) \big) \cdot \xi(t_1) = (\big( \{ 0,1 \} \big)) = Hq_2 \cdot \xi(t_2) \]

So if we computed \( P_2 \) iteratively as in (4.1) for \( q_2 = \chi_S \), then \( t_1 \) and \( t_2 \) would not be split, and we would reach the termination condition \( P_2 = Q_2 \) before all behaviourally inequivalent states have been separated.

Already Paige and Tarjan [25, Step 6 of the Algorithm] note that one additionally needs to split by \( C \setminus S = \{ c_3 \} \), which is accomplished by splitting by \( q_3 = \chi^C_S \). This is formally captured by the condition we introduce next.

\[ Hq_1 \cdot \xi(t_1) = \big( \text{id} \times \mathcal{P}\xi(Hq_0) \big) \cdot \xi(t_1) = (\big( \mathcal{P} \cdot \{ c_1, c_2 \} \big)) = Hq_1 \cdot \xi(t_2) \]

\[ Hq_2 \cdot \xi(t_1) = \big( \text{id} \times \mathcal{P}\chi_S(c_3) \big) \cdot \xi(t_1) = (\big( \{ 0,1 \} \big)) = Hq_2 \cdot \xi(t_2) \]
Lemma 4.9. For \( a : D \to A, b : D \to B \) in \( \text{Set} \), the following are equivalent:
1. \( \ker a \cup \ker b \Rightarrow D \) is a kernel.
2. \( \ker a \cup \ker b \Rightarrow D \) is the kernel of the pushout of \( a \) and \( b \).
3. For all \( x, y, z \in D \), \( ax = ay \) and \( by = bz \) implies \( ax = ay = az \) or \( bx = by = bz \).
4. For all \( x \in D \), \( [x]_a \subseteq [x]_b \) or \( [x]_b \subseteq [x]_a \).

The last item states that when going from \( a \)-equivalence classes to \( b \)-equivalence classes, the classes either merge or split, but do not merge with other classes and split at the same time. Note that in Figure 3, \( Q_1 \cup \ker \chi_S \) fails to be transitive, while \( Q_1 \cup \ker \chi_S^C \) is transitive.

Example 4.10. All \( \text{select}(X \xrightarrow{y} Y \xrightarrow{z} Z) \) routines in Theorem 3.3 respect compound blocks.

Proposition 4.11. Let \( a : D \to A, b : D \to B \) be a span such that \( \ker a \cup \ker b \) is a kernel, and let \( H : \text{Set} \to \mathcal{D} \) be a zippable functor preserving monos. Then we have
\[
\ker(Ha, Hb) = \ker H(a, b).
\] (4.2)

We thus obtain soundness of optimization (4.1); summing up:

Corollary 4.12. Suppose that \( H \) is a zippable endofunctor on \( \text{Set} \) and that \( \text{select} \) respects compound blocks and never discards all new information. Then Algorithm 3.4 with optimization (4.1) terminates and computes the simple quotient of a given finite \( H \)-coalgebra.

Remark. Like most results on set coalgebras, the above extends to multisorted sets by componentwise arguments, and this allows dealing with complex composite functors [29]. We restrict to a case with lightweight notation: Let \( F \) and \( G \) be zippable \( \text{Set} \)-functors, recalling from Theorem 4.6 that the composite \( FG \) need not itself be zippable, and let \( F \) be finitary. Then in lieu of \( FG \)-coalgebras, we can equivalently consider coalgebras for the endofunctor \( H : (X,Y) \mapsto (FY,GX) \) on \( \text{Set}^2 \). In particular, a coalgebra \( \xi : X \to FGX \) with finite carrier \( X \) induces, since \( F \) is finitary, a finite set \( Y \subseteq_t GX \), with inclusion \( y \), such that \( \xi = (X \xrightarrow{z} FY \xrightarrow{Fy} FGX) \), so we obtain a finite \( H \)-coalgebra \( (x, y) : (X,Y) \to (FY,GX) \). Mutatis mutandis, Proposition 4.11 holds also for \( H \), since kernels and pairs in \( \text{Set}^2 \) are computed componentwise, so we obtain a version of Theorem 4.12 for \( H \). Explicitly, when computing the kernel of \( Hq_{i+1} \cdot (x, y) \), we can use the optimization (4.1) in both sorts. The first component of the simple quotient of \( ((X,Y), (x, y)) \) computed by the algorithm then yields the simple quotient of the original \( (X, \xi) \). Composites of more than two functors are treated similarly.

Example 4.13. Applying this to the functors \( F = \mathcal{P}_1, G = A \times (\_), \) and \( H = \mathcal{D} \), we obtain simple (resp. general) Segala systems as coalgebras for \( FHG \) (resp. \( FHG \)). For simple Segala systems, the \( \text{Set}^3 \) functor is defined by \( (X,Y,Z) \mapsto (FY,GZ,HX) \).

5 Efficient Calculation of Kernels

In Algorithm 3.4, it is left unspecified how the kernels are computed concretely. We proceed to define a more concrete algorithm based on a refinement interface of the functor. This interface is aimed at efficient implementation of the refinement step in the algorithm. Specifically from now on, we split along \( \xi : X \to HY \) w.r.t. a subblock \( S \subseteq C \subseteq Y/Q \), and need to compute how the splitting of \( C \) into \( S \) and \( C \setminus S \) within \( Y/Q \) affects the partition \( X/P \).

The low complexity of Paige-Tarjan-style algorithms hinges on this refinement step running in time \( O(|\text{pred}[S]|) \), where \( \text{pred}(y) \) denotes the set of predecessors of some \( y \in Y \)
in the given transition system. In order to speak about “predecessors” w.r.t. more general \( \xi : X \to HY \), the refinement interface will provide an encoding of \( H \)-coalgebras as sets of states with successor states encoded as bags (implemented as lists up to ordering; recall that \( B_\ell Z \) denotes the set of bags over \( Z \)) of \( A \)-labelled edges, where \( A \) is an appropriate label alphabet. Moreover, the interface will allow us to talk about the behaviour of elements of \( X \) w.r.t. the splitting of \( C \) into \( S \) and \( C \setminus S \), looking only at points in \( S \).

**Definition 5.1.** A refinement interface for a \( \textbf{Set} \)-functor \( H \) is formed by a set \( A \) of labels, a set \( W \) of weights and functions

\[
\begin{align*}
\vartriangleright &: HY \to B_\ell(A \times Y), & \text{init} &: H^1 \times B_\ell A \to W, \\
w &: \mathcal{P} Y \to HY \to W, & \text{update} &: B_\ell A \times W \to W \times H3 \times W
\end{align*}
\]

such that for every \( S \subseteq C \subseteq Y \), the diagrams

\[
\begin{tikzcd}
H1 \times B_\ell A \arrow{r}{(\text{init})} \arrow{d}{w(Y)} & W \arrow{d}{w(w(C))} \\
H^1 \times B_\ell A \arrow{r}{(\vartriangleright,w(C))} & HY \times W \end{tikzcd}
\]

commute, where \( \text{fil}_S : B_\ell(A \times Y) \to B_\ell(A) \) is the filter function \( \text{fil}_S(f)(a) = \sum_{y \in S} f(a,y) \) for \( S \subseteq Y \). The significance of the set \( H3 \) is that when using a set \( S \subseteq C \subseteq X \) as a splitter, we want to split every block \( B \) in such a way that it becomes compatible with \( S \) and \( C \setminus S \), i.e. we group the elements \( s \in B \) by the value of \( H \chi_C^S \cdot \xi(s) \in H3 \). The set \( W \) depends on the functor. But in most cases \( W = H2 \) and \( w(C) = H \chi_C : HY \to H2 \) are sufficient.

In an implementation, we do not require a refinement interface to provide \( w \) explicitly, because the algorithm will compute the values of \( w \) incrementally using (5.1), and \( \vartriangleright \) need not be implemented because we assume the input coalgebra to be already encoded via \( \vartriangleright \):

**Definition 5.2.** Given an interface of \( H \) (Theorem 5.1), an encoding of a morphism \( \xi : X \to HY \) is given by a set \( E \) and maps

\[
\begin{align*}
\text{graph} &: E \to X \times A \times Y, & \text{type} &: X \to H^1
\end{align*}
\]

such that \( (\vartriangleright \cdot \xi(x))(a,y) = \{(e \in E \mid \text{graph}(e) = (x,a,y))\} \), and with type \( = H^1 \cdot \xi \).

Intuitively, an encoding presents the morphism \( \xi \) as a graph with edge labels from \( A \).

**Lemma 5.3.** Every morphism \( \xi : X \to HY \) has a canonical encoding where \( E \) is the obvious set of edges of \( \vartriangleright \cdot \xi : X \to B_\ell(A \times Y) \). If \( X \) is finite, then so is \( E \).

**Example 5.4.** In the following examples, we take \( W = H2 \) and \( w(C) = H \chi_C : HY \to H2 \). We use the helper function \( \text{val} := \langle H(\neq 2), \text{id}, H(\neq 1) \rangle : H3 \to H3 \times H3 \times H2 \), where \((\neq x) : 3 \to 2 \) is the equality check for \( x \in \{1,2\} \), and in each case define \( \text{update} = \text{val} \cdot \text{up} \) for some function \( \text{up} : B_\ell A \times H2 \to H3 \). We implicitly convert sets into bags.

1. For the monoid-valued functor \( C^{(-)} \), for an Abelian group \( (G,+,0) \), we take labels \( A = G \) and define \( \text{fil}(f) = \{(f(y),y) \mid y \in Y, f(y) \neq 0 \} \) (which is in finite because \( f \) is finitely supported). With \( W = H2 = G \times G \), the weight \( w(C) = H \chi_C : HY \to G \times G \) is the accumulated weight of \( Y \setminus C \) and \( C \). Then the remaining functions are

\[
\begin{align*}
\text{init}(h_1,e) &= (0,\Sigma e) & \text{up}(e,(r,c)) &= (r,c - \Sigma e, \Sigma e),
\end{align*}
\]

where \( \Sigma : B_\ell G \to G \) is the obvious summation map.
2. Similarly to the case $\mathbb{R}^{(r)}$, one has the following init and up functions for the distribution functor $\mathcal{D}$: put init$(h_1, c) = (0, 1) \in \mathcal{D} 2 \subset [0,1]^2$ and up$(e, (r, c)) = (r, c - \Sigma e, \Sigma e)$ if the latter lies in $\mathcal{D} 3$, and $(0, 0, 1)$ otherwise.

3. Similarly, one obtains a refinement interface for $\mathcal{B} l = \mathbb{N}^{(-)}$ adjusting the one for $\mathbb{Z}^{(-)}$; in fact, init remains unchanged and up$(e, (r, c)) = (r, c - \Sigma e, \Sigma e)$ if the middle component is a natural number and $(0, 0, 0)$ otherwise.

4. Given a polynomial functor $H_{\Sigma}$ for a signature $\Sigma$ with bounded arity (i.e. there exists $k$ such that every arity is at most $k$), the labels $A = \mathbb{N}$ encode the indices of the parameters:

\[
\begin{align*}
    b(\sigma(y_1, \ldots, y_n)) &= \{(1, y_1), \ldots, (n, y_n)\} & \text{init}(\sigma(0, \ldots, 0), f) &= \sigma(1, \ldots, 1) \\
    \text{up}(I, \sigma(b_1, \ldots, b_n)) &= \sigma(b_1 + (1 \in I), \ldots, b_i + (i \in I), \ldots, b_n + (n \in I))
\end{align*}
\]

Here $b_i + (i \in I)$ means $b_i + 1$ if $i \in I$ and $b_i$ otherwise. Since $I$ are the indices of the parameters in the subblock, $i \in I$ happens only if $b_i = 1$.

One example where $W = H2$ does not suffice is the powerset functor $\mathcal{P}$: Even if we know for a $t \in \mathcal{P} Y$ that it contains elements in $C \subseteq Y$, in $S \subseteq C$, and outside $C$ (i.e. we know $\mathcal{P} \chi_S(t), \mathcal{P} \chi_C \in \mathcal{P} 2$), we cannot determine whether there are any elements in $C \setminus S$ – but as seen in Theorem 4.7, we need to include this information.

**Example 5.5.** The interface for the powerset functor needs to count the edges into blocks $C \supseteq S$ in order to know whether there are edges into $C \setminus S$, as described by Paige and Tarjan [25]. What happens formally is that first the interface for $\mathbb{N}^{(-)}$ is implemented for edge weights at most 1, and then the middle component of the result of update is adjusted. So $W = \mathbb{N}^{(2)}$, and the encoding $b : \mathcal{P} Y \rightarrow \mathcal{B} l(1 \times Y)$ is the obvious inclusion. Then

\[
\begin{align*}
    \text{init}(h_1, c) &= (0, |e|) & \text{w}(C)(M) &= \mathcal{B} l \chi_C(M) = ([M \setminus C], |C \cap M|) \\
    \text{update}(n, (c, r)) &= (\mathcal{B} l = 2, (> 0)^3, \mathcal{B} l = 1))(r, c - |n|, |n|) \\
    &= ((r + c - |n|, |n|), (r > 0, c - |n| > 0, |n| > 0), (r + |n|, c - |n|)),
\end{align*}
\]

where $x > 0$ is 0 if $x = 0$ and 1 otherwise.

**Assumption 5.6.** From now on, assume a Set-functor $H$ with a refinement interface such that init and update run in linear time and elements of $H3$ can be compared in constant time.

**Example 5.7.** The refinement interfaces in Examples 5.4 and 5.5 satisfy Assumption 5.6.

**Remark.** In the implementation, we encode the partitions $X/P, Y/Q$ as doubly linked lists of the blocks they contain, and each block is in turn encoded as a doubly linked list of its elements. The elements $x \in X$ and $y \in Y$ each hold a pointer to the corresponding list entry in the blocks containing them. This allows removing elements from a block in $O(1)$.

The algorithm maintains the following mutable data structures:

- An array toSub : $X \rightarrow \mathcal{B} l E$, mapping $x \in X$ to its outgoing edges ending in the currently processed subblock.
- A pointer mapping edges to memory addresses: lastW : $E \rightarrow \mathbb{N}$.
- A store of last values deref : $\mathbb{N} \rightarrow W$.
- For each block $B$ a list of markings mark$_B \subseteq B \times \mathbb{N}$.

**Notation 5.8.** In the following we write $e = x \xrightarrow{a} y$ in lieu of graph$(e) = (x, a, y)$. 

32:12  Efficient Coalgebraic Partition Refinement
1. for e ∈ E, e = x \xrightarrow{a} y do
2. add e to toSub[x] and pred[x].
3. for x ∈ X do
4. pX := new cell in deref containing init(type[x], B(\pi_2 \cdot graph)(toSub[x]))
5. for e ∈ toSub[x] do lastW[e] = pX
6. toSub[x] := ∅
7. X/P := group X by type : X → H1, Y/Q := {Y}.

Figure 4 The initialization procedure.

Definition 5.9 (Invariants). Our correctness proof below establishes the following properties that we call the invariants:
1. For all x ∈ X, toSub(x) = ∅, i.e. toSub is empty everywhere.
2. For e_i = x_i \xrightarrow{a_i} y_i, i ∈ \{1, 2\}, lastW(e_1) = lastW(e_2) ⇔ x_1 = x_2 and [y_1]_{\kappa_Q} = [y_2]_{\kappa_Q}.
3. For each e = x \xrightarrow{a} y, C := [y]_{\kappa_Q} ∈ Y/Q, w(C, \xi(x)) = deref \cdot lastW(e).
4. For x_1, x_2 ∈ B ∈ X/P, C ∈ Y/Q, (x_1, x_2) ∈ ker(H_{\chi \cdot \xi}).

In the following code listings, we use square brackets for array lookups and updates in order to emphasize they run in constant time. We assume that the functions graph : E → X ∗ A ∗ Y and type : X → H1 are implemented as arrays. In the initialization step the predecessor array pred : Y → \bigcap_i E, pred(y) = \{e ∈ E \mid e = x \xrightarrow{a} y\} is computed. Sets and bags are implemented as lists. We only insert elements into sets not yet containing them.

We say that we group a finite set Z by f : Z → Z’ to indicate that we compute \lfloor - \rfloor_f. This is done by sorting the elements of z ∈ Z by a binary encoding of f(z) using any \mathcal{O}(\lfloor Z \rfloor \cdot \log \lfloor Z \rfloor) sorting algorithm, and then grouping elements with the same f(z) into blocks. In order to keep the overall complexity for the grouping operations low enough, one needs to use a possible majority candidate during sorting, following Valmari and Franceschinis [32].

The algorithm computing the initial partition is listed in Figure 4.

Lemma 5.10. The initialization procedure runs in time \mathcal{O}(\lfloor E \rfloor + \lfloor X \rfloor \cdot \log \lfloor X \rfloor) and makes the invariants true.

The algorithm for one refinement step along a morphism ξ : X → HY is listed in Figure 5.

In the first part, all blocks B ∈ X/P are collected that have an edge into S, together with v_\varnothing ∈ H3 which represents H_{X_S}^C : ξ(x) for any x ∈ B that has no edge into S. For each x ∈ X, toSub[x] collects the edges from x into S. The markings mark_B list those elements x ∈ B that have an edge into S, together with a pointer to w(C, x).

In the second part, each block B with an edge into S is refined w.r.t. H_{X_S}^C : ξ. First, for any (x, p_C) ∈ mark_B, we compute w(S, x), v^x = H_{X_S}^C : ξ(x), and w(C \setminus S, x) using update. Then, the weight of all edges x → C \setminus S is updated to w(C \setminus S, x) and the weight of all edges x → S needs to be stored in a new cell containing w(S, x). For all unmarked x ∈ B, we know that H_{X_S}^C : ξ(x) = v_\varnothing; so all x with v^x = v_\varnothing stay in B. All other x ∈ B are removed and distributed to new blocks w.r.t. v^x.

Theorem 5.11. Split(X/P, Y/Q, S ≤ C ∈ Y/Q) refines X/P by H_{X_S}^C : ξ : X → H3.

Lemma 5.12. After running Split, the invariants hold.

Lemma 5.13. Lines 1–23 in Split run in time \mathcal{O}(\sum_{y ∈ S} \lfloor pred(y) \rfloor).
Efficient Coalgebraic Partition Refinement

\begin{figure}[h]
\centering
\begin{minipage}[t]{0.45\textwidth}
\begin{algorithmic}
\State \textbf{SPLIT}(X/P, Y/Q, S \subseteq C \in Y/Q)
\State $M := \emptyset \times X/P \times H3$
\State $\text{for } y \in S, e \in \text{pred}[y] \text{ do}$
\State $x \rightarrow y; e$
\State $B := \text{block with } x \in B \in X/P$
\State \textbf{if} mark$_B$ is empty \textbf{then}
\State $w_C := \text{deref} \cdot \text{lastW}[e]$
\State $v_\emptyset := \pi_2 \cdot \text{update}(\emptyset, w_C)$
\State $\text{add } (B, v_\emptyset) \text{ to } M$
\State \textbf{if} toSub[$x$] = $\emptyset$ \textbf{then}
\State $\text{add } (x, \text{lastW}[e]) \text{ to } \text{mark}_B$
\State $\text{add } e \text{ to } \text{toSub}[x]$
\end{algorithmic}
\caption{Collecting predecessor blocks}
\end{minipage}
\begin{minipage}[t]{0.45\textwidth}
\begin{algorithmic}
\For{$(B, v_\emptyset) \in M$}
\State $B_{\#} := \emptyset \times X \times H3$
\For{$(x, p_C) \in \text{mark}_B$}
\State $\ell := B_\emptyset(\pi_2 \cdot \text{graph}(\text{toSub}[x]))$
\State $(w_S, \emptyset, \emptyset) := \text{update}(\ell, \text{deref}[p_C])$
\State $\text{deref}[p_C] := \emptyset_{C/S}$
\State $p_S := \text{new cell containing } w_S$
\State $\text{for } e \in \text{toSub}[x] \text{ do } \text{lastW}[e] := p_S$
\State $\text{toSub}[x] := \emptyset$
\EndFor
\If{$v_x \neq v_\emptyset$}
\State remove $x$ from $B$
\State insert $(x, v_x)$ into $B_{\#}$
\EndIf
\State $B_1 \times \{v_1\}, \ldots, B_\ell \times \{v_\ell\} :=$ group $B_{\#}$ by $\pi_2 : X \times H3 \rightarrow H3$
\State $\text{insert } B_1, \ldots, B_\ell := \text{into } X/P$
\EndFor
\caption{Splitting predecessor blocks}
\end{algorithmic}
\end{minipage}
\end{figure}

\textbf{Lemma 5.14.} For $S_i \subseteq C_i \in Y/Q_i, 0 \leq i < k$, with $2 \cdot |S_i| \leq |C_i|$ and $Q_{i+1} = \ker(\kappa_{Q_i} : \chi_{S_i})$:
\begin{enumerate}
\item For each $y \in Y$, $|\{i < k \mid y \in S_i\}| \leq \log_2 |Y|$.\end{enumerate}
\begin{enumerate}
\item $\text{SPLIT}(S_i \subseteq C_i \in Y/Q_i)$ for all $0 \leq i < k$ takes at most $O(|E| \cdot \log |Y|)$ time in total.
\end{enumerate}

Bringing Sections 3, 4, and 5 together, take a coalgebra $\xi : X \rightarrow HX$ for a zippable Set-functor $H$ with a given refinement interface where \text{init} and \text{update} run in linear and comparison in constant time. Instantiate Algorithm 3.4 with the select routine from Theorem 3.3.1, making $q_{i+1} = k_{i+1} \cdot \kappa_B = \chi_{S_i}$ with $2 \cdot |S| \leq |C|$, $S_i, C_i \subseteq X$. Replace line 4 by
\[ X/P_{i+1} = \text{SPLIT}(X/P_i, X/Q_i, S_i \subseteq C_i). \tag{4.1} \]

By Theorem 5.11 this is equivalent to (4.1), and hence, by Theorem 4.12, to the original line 4, since $\chi_{S_i}$ respects compound blocks. By Lemmas 5.10 and 5.14.2, we have

\textbf{Theorem 5.15.} The above instance of Algorithm 3.4 computes the quotient modulo behavioural equivalence of a given coalgebra in time $O((m + n) \cdot \log n)$, for $m = |E|$, $n = |X|$.\vspace{0.5cm}

\textbf{Example 5.16.}\vspace{0.5cm}
\begin{enumerate}
\item For $H = I \times P_I$, we obtain the classical Paige-Tarjan algorithm [25] (with initial partition $I$), with the same complexity $O((m + n) \cdot \log n)$.
\item For $HX = I \times \mathbb{R}^X$, we solve Markov chain lumping with an initial partition $I$ in time $O((m + n) \cdot \log n)$, like the best known algorithm (Valmari and Franceschinis [32]).
\item For infinite $A$, we need to decompose the functor $P_A(A \times (-))$ for labelled transition systems into $P_I$ and $A \times (-)$, and thus obtain run time $O(m \cdot \log m)$, like in [10] but slower than Valmari’s $O(m \cdot \log n)$ [31]. For fixed finite $A$, we run in time $O(m \cdot \log n)$.
\item Hopcroft’s classical automata minimization [14] is obtained by $HX = 2 \times X^A$, with running time $O(n \cdot \log n)$ for fixed alphabet $A$. For non-fixed $A$ the best known complexity is $O(|A| \cdot \log n)$ [12, 21]. By using decomposition into $2 \times P_I$ and $I \times (-)$ we obtain $O(|A| \cdot n \cdot \log n + |A| \cdot n \cdot \log |A|)$.
\item We quotient simple (resp. general) Segala systems [30] by bisimilarity after decomposition into three sorts (cf. Theorem 4.13). The time bound $O((n + m) \log(n + m))$ slightly improves on the previous bound [3].
\end{enumerate}
6 Conclusions and Future Work

We have presented a generic algorithm that quotients coalgebras by behavioural equivalence. We have started from a category-theoretic procedure that works for every mono-preserving functor on a category with image factorizations, and have then developed an improved algorithm for **zippable** endofunctors on $\text{Set}$. Provided the given type functor can be equipped with an efficient implementation of a refinement interface, we have finally arrived at a concrete procedure that runs in time $O((m + n) \log n)$ where $m$ is the number of edges and $n$ the number of nodes in a graph-based representation of the input coalgebra. We have shown that this instantiates to (minor variants of) several known efficient partition refinement algorithms: the classical Hopcroft algorithm [14] for minimization of DFAs, the Paige-Tarjan algorithm for unlabelled transition systems [25], and Valmari and Franceschinis’s lumping algorithm for weighted transition systems [32]. Moreover, we obtain a new algorithm for simple Segala systems that is asymptotically faster than previous algorithms [3]. Coverage of Segala systems is based on modularity results in multi-sorted coalgebra [29].

It remains open whether our approach can be extended to, e.g., the monotone neighbourhood functor, which is not itself zippable and also does not have an obvious factorization into zippable functors. We do expect that our algorithm applies beyond weighted systems.

References


The Complexity of Flat Freeze LTL

Benedikt Bollig¹, Karin Quaas², and Arnaud Sangnier³

1 CNRS, LSV, ENS Paris-Saclay, Université Paris-Saclay, France
2 Universität Leipzig, Germany
3 IRIF, Université Paris Diderot, France

Abstract

We consider the model-checking problem for freeze LTL on one-counter automata (OCAs). Freeze LTL extends LTL with the freeze quantifier, which allows one to store different counter values of a run in registers so that they can be compared with one another. As the model-checking problem is undecidable in general, we focus on the flat fragment of freeze LTL, in which the usage of the freeze quantifier is restricted. Recently, Lechner et al. showed that model checking for flat freeze LTL on OCAs with binary encoding of counter updates is decidable and in \(2NEXP\). In this paper, we prove that the problem is, in fact, \(NEXP\)-complete no matter whether counter updates are encoded in unary or binary. Like Lechner et al., we rely on a reduction to the reachability problem in OCAs with parameterized tests (OCAPs). The new aspect is that we simulate OCAPs by alternating two-way automata over words. This implies an exponential upper bound on the parameter values that we exploit towards an \(NP\) algorithm for reachability in OCAPs with unary updates. We obtain our main result as a corollary.

1998 ACM Subject Classification F.1.1 Models of Computation

Keywords and phrases one-counter automata, freeze LTL, model checking

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2017.33

1 Introduction

One-counter automata (OCAs) are a simple yet fundamental computational model operating on a single counter that ranges over the non-negative integers. Albeit being a classical model, OCAs are in the focus of ongoing research in the verification community (cf., for example, [22, 19, 26, 9, 18, 20, 23]). A large body of work is devoted to model checking OCAs, i.e., the question whether all runs of a given OCA satisfy a temporal-logic specification. A natural way to model runs of OCAs is in terms of data words, i.e., words where every position carries two pieces of information: a set of propositions from a finite alphabet, and a datum from an infinite alphabet. In our case, the datum represents the current counter value. Now, reasoning about the sequence of propositions that is produced by a run amounts to model checking OCAs against classical temporal logics like linear-time temporal logic (LTL) and computation-tree logic (CTL) [20, 19, 18, 21]. But it is natural to also reason about the unboundedly many counter values that may occur. Several formalisms have been introduced that can handle infinite alphabets, including variants or extensions of monadic second-order logic, LTL, and CTL [10, 12, 3, 15]. Freeze LTL is an extension of LTL that allows one to remember, in terms of the freeze quantifier, certain counter values for later comparison in a run of the OCA under consideration (cf. [16, 27, 11, 10]). Unfortunately, satisfiability and model checking OCAs against freeze LTL are undecidable [10, 12].
In this paper, we study model checking of OCAs against formulas in flat freeze LTL, a fragment of freeze LTL that restricts the freeze quantifier, but allows for unlimited usage of comparisons. A typical property definable in flat freeze LTL is “there exists a counter value that occurs infinitely often”. Moreover, the negation of many natural freeze LTL specifications can be expressed in flat freeze LTL. The approach of restricting the syntax of a temporal logic to its flat fragment has also been pursued in [7] for Constraint LTL, and in [4] for MTL.

Demri and Sangnier [13] reduced model checking OCAs against flat freeze LTL to reachability in OCAs with parameterized tests (OCAPs). In an OCAP, counter values may be compared with parameters whose values are arbitrary but fixed. Reachability asks whether a given state can be reached under some parameter instantiation. Essentially, the translation of a flat freeze LTL formula into an OCAP interprets every freeze quantifier as a parameter, whose value can be compared with counter values arbitrarily often. Decidability of the reachability problem for OCAPs, however, was left open. Recently, Lechner et al. proved decidability by a reduction to satisfiability in Presburger arithmetic [26]. As a corollary, they obtain a 2NEXPTIME upper bound for model checking OCAs against formulas in flat freeze LTL, assuming that counter updates in OCAs are encoded in binary.

Our main technical contribution is an improvement of the result by Lechner et al. We proceed in two main steps. First, we show that reachability for OCAPs (with unary counter updates) can be reduced to non-emptiness of alternating two-way (finite) automata. Interestingly, alternating two-way automata have already been used as an algorithmic framework for game-based versions of pushdown processes [24, 6], one-counter processes [28], and timed systems [1]. Our link already implies decidability of both reachability for OCAPs (in PSPACE when counter updates are unary) and model checking OCAs against flat freeze LTL (in EXPSPACE). However, we can go further. First, we deduce an exponential upper bound on the largest parameter value needed for an accepting run in the given OCAP. Exploiting this bound and a technique by Galil [17], we show that, actually, reachability for OCAPs is in NP. As a corollary, we obtain a NEXPTIME upper bound for model checking OCAs against flat freeze LTL. Using a result from [18], we can also show NEXPTIME-hardness, which establishes the precise complexity of the model-checking problem. Our result applies no matter whether counter updates in the given OCA are encoded in unary or binary.

Outline. In Section 2, we define OCAs, (flat) freeze LTL, and the corresponding model-checking problems. Section 3 is devoted to reachability in OCAPs, which is at the heart of our model-checking procedures. The reduction of model checking to reachability in OCAPs is given in Section 4, where we also present lower bounds. We conclude in Section 5.

2 Preliminaries

2.1 One-Counter Automata with Parameterized Tests

We start by defining one-counter automata with all extras such as parameters and succinct encodings of updates. Well-known subclasses are then identified as special cases.

For the rest of this paper, we fix a countably infinite set \( \mathbb{P} \) of propositions, which will label states of a one-counter automaton. Transitions of an automaton may perform tests to compare the current counter value with zero or with a parameter. Thus, for a finite set \( X \) of parameters, we let \( \text{Tests}(X) = \{ \text{zero}\} \cup \{<x, =x, >x \mid x \in X\} \).

Definition 1. A succinct one-counter automaton with parameterized tests (SOCAP) is a tuple \( \mathcal{A} = (Q, X, q_\text{in}, \Delta, \mu) \) where \( Q \) is a finite set of states, \( X \) is a finite set of parameters
ranging over \( \mathbb{N} \), \( q_n \in Q \) is the initial state, \( \Delta \subseteq Q \times (\mathbb{Z} \cup \text{Tests}(X)) \times Q \) is a finite set of transitions, and \( \mu : Q \to 2^Z \) maps each state to a finite set of propositions. We define the size of \( A \) as \( |A| = |Q| + |X| + \sum_{q \in Q} |\mu(q)| + |\Delta| \cdot \max\{1\} \cup \{\log(|z|) \mid (q,z,q') \in \Delta \cap (Q \times (\mathbb{Z} \setminus \{0\}) \times Q)\} \).

Thus, transitions of a SOCAP either compare the counter value with zero or a parameter value, or increment/decrement the counter by a value \( z \in \mathbb{Z} \), which is encoded in binary.

Let \( \mathcal{C}_A := Q \times \mathbb{N} \) be the set of configurations of \( A \). In a configuration \( (q,v) \in \mathcal{C}_A \), the first component \( q \) is the current state and \( v \) is the current counter value which is always non-negative. The semantics of \( A \) is given w.r.t. a parameter instantiation \( \gamma : X \to \mathbb{N} \) in terms of a global transition relation \( \rightarrow_{\gamma} \subseteq \mathcal{C}_A \times \mathcal{C}_A \). For two configurations \( (q,v) \) and \( (q',v') \), we have \( (q,v) \rightarrow_{\gamma} (q',v') \) if there is \( (q,op,q') \in \Delta \) such that one of the following holds:

- \( op \in \mathbb{Z} \) and \( v' = v + op \),
- \( op = \text{zero} \) and \( v = v' = 0 \), or
- \( op = \Box x \) and \( v = v' \) and \( v \trianglerighteq \gamma(x) \), for some \( \Delta \in \{<,\leq,\geq\} \).

A \( \gamma \)-run of \( A \) is a finite or infinite sequence \( \rho = (q_0,v_0) \rightarrow_{\gamma} (q_1,v_1) \rightarrow_{\gamma} \cdots \) of global transitions (a run may consist of one single configuration \( (q_0,v_0) \)). We say that \( \rho \) is initialized if \( q_0 = q_n \) and \( v_0 = 0 \).

We identify some well-known special cases of the general model. Let \( A = (Q,X,q_n,\Delta,\mu) \) be a SOCAP. We say that

- \( A \) is a one-counter automaton with parameterized tests (OCAP) if all transition labels are among \( \{+1,0,-1\} \cup \text{Tests}(X) \) (i.e., counter updates are unary);
- \( A \) is a succinct one-counter automaton (SOCIA) if \( X = \emptyset \);
- \( A \) is a one-counter automaton (OCA) if \( X = \emptyset \) and, moreover, all transition labels are among \( \{+1,0,-1\} \cup \{\text{zero}\} \).

Note that, when \( X = \emptyset \) (i.e., in the case of a SOCA or OCA), we may omit \( X \) and simply refer to \( (Q,q_n,\Delta,\mu) \). Also, the global transition relation does not depend on a parameter instantiation anymore so that we may just write \( \rightarrow \) instead of \( \rightarrow_{\gamma} \). Similarly, for reachability problems (see below), \( \mu \) is irrelevant so that it can be omitted, too.

One of the most fundamental decision problems we can consider is whether a given state \( q_f \in Q \) is reachable. Given some parameter instantiation \( \gamma \), we say that \( q_f \) is \( \gamma \)-reachable if there is an initialized run \( (q_0,v_0) \rightarrow_{\gamma} \cdots \rightarrow_{\gamma} (q_n,v_n) \) such that \( q_n = q_f \). Moreover, \( q_f \) is reachable, written \( A \models \text{Reach}(q_f) \), if it is \( \gamma \)-reachable for some \( \gamma \). Now, for a class \( \mathcal{C} \in \{\text{SOCAP}, \text{OCAP}, \text{SOCA}, \text{OCA}\} \), the reachability problem is defined as follows:

\[
\textbf{C-Reachability} \\
\text{Input: } A = (Q,X,q_n,\Delta) \in \mathcal{C} \text{ and } q_f \in Q \\
\text{Question: } \text{Do we have } A \models \text{Reach}(q_f) ?
\]

Towards the Büchi problem (or repeated reachability), we write \( A \models \text{Reach}^\omega(q_f) \) if there are a parameter instantiation \( \gamma \) and an infinite initialized \( \gamma \)-run \( (q_0,v_0) \rightarrow_{\gamma} (q_1,v_1) \rightarrow_{\gamma} \cdots \) such that \( q_i = q_f \) for infinitely many \( i \in \mathbb{N} \). The Büchi problem asks whether some state
from a given set $F \subseteq Q$ can be visited infinitely often:

<table>
<thead>
<tr>
<th>C-Büchi</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> $A = (Q, X, q_0, \Delta) \in C$ and $F \subseteq Q$</td>
</tr>
<tr>
<td><strong>Question:</strong> Do we have $A \models \text{Reach}^\omega(q_f)$ for some $q_f \in F$?</td>
</tr>
</tbody>
</table>

It is known that OCA-Reachability and OCA-Büchi are NLOGSPACE-complete [29, 9], and that SOCA-Reachability and SOCA-Büchi are NP-complete [22, 21] (cf. also Table 1).

## 2.2 Freeze LTL and Its Flat Fragment

We now define freeze LTL. To do so, we fix a countably infinite supply of registers $R$.

> **Definition 2 (Freeze LTL).** The logic freeze LTL, denoted by $\text{LTL}^\dagger$, is given by the grammar

$$\varphi ::= p \mid r > r \mid \neg \varphi \mid \varphi \lor \varphi \mid \varphi \land \varphi \mid X\varphi \mid \varphi U \varphi \mid \varphi R \varphi \mid \downarrow_r \varphi$$

where $p \in \mathbb{P}$, $r \in R$, and $\mathbb{R} \in \{<,=,>\}$.

Note that, apart from the until operator $U$, we also include the dual release operator $R$. This is convenient in proofs, as it allows us to assume formulas to be in negation normal form. We also use common abbreviations such as $\varphi \rightarrow \psi$ for $\neg \varphi \lor \psi$, $F \varphi$ for $\text{true} U \varphi$ (with $\text{true} = p \lor \neg p$ for some proposition $p$), and $G \varphi$ for $\neg F \neg \varphi$.

We call $\downarrow_r$ the freeze quantifier. It stores the current counter value in register $r$. Atomic formulas of the form $\varphi \lor r$, called register tests, perform a comparison of the current counter value with the contents of $r$, provided that they are in the scope of a freeze quantifier. Actually, inequalities of the form $<r$ and $>r$ were not present in the original definition of LTL$^\dagger$, but our techniques will take care of them without extra cost or additional technical complication. Classical LTL is obtained as the fragment of LTL$^\dagger$ that uses neither the freeze quantifier nor register tests.

A formula $\varphi \in \text{LTL}^\dagger$ is interpreted over an infinite sequence $w = (P_0, v_0)(P_1, v_1) \ldots \in (2^P \times \mathbb{N})^\omega$ with respect to a position $i \in \mathbb{N}$ and a register assignment $\nu : R \rightarrow \mathbb{N}$. The satisfaction relation $w, i \models_{\nu} \varphi$ is inductively defined as follows:

- $w, i \models_{\nu} p$ if $p \in P_i$,
- $w, i \models_{\nu} r > r$ if $v_i \succ_{\nu} v(r)$,
- $w, i \models_{\nu} X\varphi$ if $w, i + 1 \models_{\nu} \varphi$,
- $w, i \models_{\nu} \varphi_1 U \varphi_2$ if there is $j \geq i$ such that $w, j \models_{\nu} \varphi_2$ and $w, k \models_{\nu} \varphi_1$ for all $k \in \{i, \ldots, j - 1\}$,
- $w, i \models_{\nu} \varphi_1 R \varphi_2$ if one of the following holds: (i) $w, k \models_{\nu} \varphi_2$ for all $k \geq i$, or (ii) there is $j \geq i$ such that $w, j \models_{\nu} \varphi_1$ and $w, k \models_{\nu} \varphi_2$ for all $k \in \{i, \ldots, j\}$,
- $w, i \models_{\nu} \downarrow_r \varphi$ if $w, i \models_{\nu[r \mapsto v_i]} \varphi$, where the register assignment $\nu[r \mapsto v_i]$ maps $r$ to $v_i$ and coincides with $\nu$ on all other registers.

Negation, disjunction, and conjunction are interpreted as usual.

Let $\varphi$ be a sentence, i.e., every subformula of $\varphi$ of the form $\varphi \lor r$ is in the scope of a freeze quantifier $\downarrow_r$. Then, the initial register assignment is irrelevant, and we simply write $w \models \varphi$ if $w, 0 \models_{\nu} \varphi$ (with $\nu$ arbitrary). Let $A = (Q, X, q_0, \Delta, \mu)$ be a SOCAP and let $\rho = (q_0, v_0) \rightarrow_\gamma (q_1, v_1) \rightarrow_\gamma \cdots$ be an infinite run of $A$. We say that $\rho$ satisfies $\varphi$, written $\rho \models \varphi$, if $(\mu(q_0), v_0)(\mu(q_1), v_1) \cdots \models \varphi$. Moreover, we write $A \models_\exists \varphi$ if there exist $\gamma$ and an infinite initialized $\gamma$-run $\rho$ of $A$ such that $\rho \models \varphi$. 

Table 1 Old and new results.

<table>
<thead>
<tr>
<th></th>
<th>Reachability/ Büchi</th>
<th>MC(LTL)</th>
<th>MC(flatLTL↓)</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCA</td>
<td>NLOGSPACE-complete</td>
<td>PSPECOMEplete</td>
<td>NEXPTIME-complete</td>
</tr>
<tr>
<td></td>
<td>[29] /[9]</td>
<td>(e.g., [21])</td>
<td>Theorem 21</td>
</tr>
<tr>
<td>SOCA</td>
<td>NP-complete</td>
<td>PSPECOMEplete</td>
<td>NEXPTIME-complete</td>
</tr>
<tr>
<td></td>
<td>[22] /[21]</td>
<td>[18]</td>
<td>Theorem 21</td>
</tr>
</tbody>
</table>

Model checking for a class $C$ of SOCAPs and a logic $\mathcal{L} \subseteq \text{LTL}^↓$ is defined as follows:

- **Input:** $A \in C$ and a sentence $\varphi \in \mathcal{L}$
- **Question:** Do we have $A \models \exists \varphi$?

Note that, following [12, 26], we study the existential version of the model-checking problem (“Is there some run satisfying the formula?”). The reason is that the flat fragment that we consider next is not closed under negation, and the negation of many useful freeze LTL formulas are actually flat (cf. Example 4 below).

Unfortunately, OCA-MC(flatLTL↓) is undecidable [12], even if formulas use only one register. This motivates the study of the flat fragment of LTL↓, restricting the usage of the freeze quantifier [12]. Essentially, it is no longer possible to overwrite a register unboundedly often.

**Definition 3 (Flat Freeze LTL).** The fragment flatLTL↓ of LTL↓ contains a formula $\varphi$ if, for every occurrence of a subformula $\psi = \varphi_1 U \varphi_2$ (respectively, $\psi = \varphi_2 R \varphi_1$) in $\varphi$, the following hold: (i) If the occurrence of $\psi$ is in the scope of an even number of negations, then the freeze quantifier does not occur in $\varphi_1$, and (ii) if it is in the scope of an odd number of negations, then the freeze quantifier does not occur in $\varphi_2$.

**Example 4.** An example flatLTL↓ formula is $F \downarrow_r G (\langle r \rangle \lor (r = \top))$, saying that a run takes only finitely many different counter values. The formula $\varphi = G \downarrow_r (\text{req} \rightarrow F(\text{serve} \land (r = \top)))$ (from [26]) says that every request is eventually served with the same ticket. Note that $\varphi$ is not in flatLTL↓, but its negation $\neg (\text{true} U \neg \downarrow_r (\text{req} \rightarrow F(\text{serve} \land (r = \top))))$ is: The until lies in the scope of an even number of negations, and the freeze quantifier is in the second argument of the until.

In [13], it has been shown that OCA-MC(flatLTL↓) can be reduced to OCAP-Büchi, but decidability of the latter was left open (positive results were only obtained for a restricted fragment of flatLTL↓). Recently, Lechner et al. showed that OCAP-Büchi is decidable [26]. In fact, they establish that SOCA-MC(flatLTL↓) is in $2\text{NEXPTIME}$.

Our main result states that the problems OCA-MC(flatLTL↓) and SOCA-MC(flatLTL↓) are both $\text{NEXPTIME}$-complete. A comparison with known results can be found in Table 1. The proof outline is depicted in Figure 1. Essentially, we also rely on a reduction of SOCA-MC(flatLTL↓) to OCAP-Reachability, and the main challenge is to establish the precise complexity of the latter. In Section 3, we reduce OCAP-Reachability to non-emptiness of alternating two-way automata over infinite words, which is then exploited to prove...
an NP upper bound. The reductions from SOCA-MC(flatLTL↓) to OCAP-Reachability themselves are given in Section 4, which also contains our main results.

### 3 OCAP-Reachability is NP-complete

In this section, we reduce OCAP-Reachability to non-emptiness of alternating two-way automata (A2As) over infinite words. While this already implies that OCAP-Reachability is in PSPACE, we then go a step further and show that the problem is in NP.

#### 3.1 From OCAPs to Alternating Two-Way Automata

We fix an OCAP \( \mathcal{A} = (Q, \mathcal{X}, q_0, \Delta) \). The main idea is to encode a parameter instantiation \( \gamma : \mathcal{X} \to \mathbb{N} \) as a word over the alphabet \( \Sigma = \mathcal{X} \cup \{\square\} \), where \( \square \) is a fresh symbol. A word \( w = a_0a_1a_2 \cdots \in \Sigma^\omega \), with \( a_i \in \Sigma \), is called a parameter word (over \( \mathcal{X} \)) if \( a_0 = \square \) and, for all \( x \in \mathcal{X} \), there is exactly one position \( i \in \mathbb{N} \) such that \( a_i = x \). In other words, \( w \) starts with \( \square \) and every parameter occurs exactly once. Then, \( w \) determines a parameter instantiation \( \gamma_w : \mathcal{X} \to \mathbb{N} \) as follows: If \( x = a_i \), then \( \gamma_w(x) = |a_1 \cdots a_{i-1}\square| \) where \( |a_1 \cdots a_{i-1}\square| \) denotes the number of occurrences of \( \square \) in \( a_1 \cdots a_{i-1} \) (note that we start at the second position of \( w \)). For example, given \( \mathcal{X} = \{x_1, x_2, x_3\} \), both \( w = \square x_2 \square x_1 x_3 \) and \( w' = \square x_2 \square x_3 x_1 \square \) are parameter words with \( \gamma_w = \gamma_{w'} = \{x_1 \mapsto 2, x_2 \mapsto 0, x_3 \mapsto 2\} \). Note that, for every parameter instantiation \( \gamma \), there is at least one parameter word \( w \) such that \( \gamma_w = \gamma \). Let \( \mathcal{W}_\mathcal{X} \) denote the set of all parameter words over \( \mathcal{X} \).

From \( \mathcal{A} \) and a state \( q_f \in Q \), we will build an A2A that accepts the set of parameter words \( w \) such that \( q_f \) is \( \gamma_w \)-reachable. Like a Turing machine, an A2A can read a letter, change its state, and move its head to the left or to the right (or stay at the current position). In addition, it can spawn several independent copies, for example one that goes to the left and one that goes to the right. However, unlike a Turing machine, an A2A is not allowed
to modify the input word so that its expressive power does not go beyond finite (Büchi) automata. The simulation proceeds as follows. When the OCAP increments its counter, the A2A moves to the right to the next occurrence of \( \square \). To simulate a decrement, it moves to the left until it encounters the previous \( \square \). To mimic the zero test, it verifies that it is currently on the first position of the word. Moreover, it will make use of the letters in \( X \) to simulate parameter tests. At the beginning of an execution, the A2A spawns independent copies that check whether the input word is a valid parameter word.

Let us define A2As and formalize the simulation of an OCAP. Given a finite set \( Y \), we denote by \( B^+(Y) \) the set of positive Boolean formulas over \( Y \), including \textit{true} and \textit{false}. A subset \( Y' \subseteq Y \) satisfies \( \beta \in B^+(Y) \), written \( Y' \models \beta \), if \( \beta \) is evaluated to \textit{true} when assigning \textit{true} to every variable in \( Y' \), and \textit{false} to every variable in \( Y \setminus Y' \). In particular, we have \( \emptyset \models \textit{true} \). For \( \beta \in B^+(Y) \), let \( |\beta| \) denote the size of \( \beta \), defined inductively by \( |\beta_1 \lor \beta_2| = |\beta_1 \land \beta_2| = |\beta_1| + |\beta_2| + 1 \) and \( |\beta| = 1 \) for atomic formulas \( \beta \).

**Definition 5.** An alternating two-way automaton (A2A) is a tuple \( \mathcal{T} = (S, \Sigma, s_m, \delta, S_f) \), where \( S \) is a finite set of states, \( \Sigma \) is a finite alphabet, \( s_m \in S \) is the initial state, \( S_f \subseteq S \) is the set of accepting states, and \( \delta \subseteq S \times (\Sigma \cup \{ \text{first}^? \}) \times B^+(S \times \{ +1, 0, -1 \}) \) is the finite transition relation\(^1\). A transition \((s, \text{test}, \beta) \in \delta\) will also be written \( s \xrightarrow{\text{test}} \beta \). The size of \( \mathcal{T} \) is defined as \( |\mathcal{T}| = |S| + |\Sigma| + \sum_{(s, \text{test}, \beta) \in \delta} |\beta| \).

While, in an OCAP, \( +1 \) and \( -1 \) are interpreted as \textit{increment} and \textit{decrement} the counter, respectively, their interpretation in an A2A is \textit{go to the right} and \textit{go to the left} in the input word. Moreover, 0 means \textit{stay}. Actually, when \( \delta \subseteq S \times \Sigma \times (S \times \{ +1 \}) \), then we deal with a classical Büchi automaton.

A run of \( \mathcal{T} \) on an infinite word \( w = a_0a_1a_2 \cdots \in \Sigma^\omega \) is a rooted tree (possibly infinite, but finitely branching) whose vertices are labeled with elements in \( S \times \mathbb{N} \). A node with label \((s, n)\) represents a proof obligation that has to be fulfilled starting from state \( s \) and position \( n \) in the input word. The root of a run is labeled by \((s_m, 0)\). Moreover, we require that, for every vertex labeled by \((s, n)\) with \( k \in \mathbb{N} \) children labeled by \((s_1, n_1), \ldots, (s_k, n_k)\), there is a transition \((s, \text{test}, \beta) \in \delta\) such that (i) the set \( \{(s_1, n_1 - n), \ldots, (s_k, n_k - n)\} \subseteq S \times \{ +1, 0, -1 \} \) satisfies \( \beta \), (ii) \( \text{test} = \text{first}^? \) implies \( n = 0 \), and (iii) \( \text{test} \in \Sigma \) implies \( a_n = \text{test} \). Note that, similarly to an OCAP, a transition with move \( -1 \) is blocked if \( n = 0 \), i.e., if \( \mathcal{T} \) is at the first position of the input word. A run is \textit{accepting} if every infinite branch visits some accepting state from \( S_f \) infinitely often. The language of \( \mathcal{T} \) is defined as \( L(\mathcal{T}) = \{ w \in \Sigma^\omega \mid \text{there exists an accepting run of } \mathcal{T} \text{ on } w \} \). The \textit{non-emptiness problem} for A2As is to decide, given an A2A \( \mathcal{T} \), whether \( L(\mathcal{T}) \neq \emptyset \).

**Theorem 6** ([28]). The \textit{non-emptiness problem} for A2As is in \textit{PSPACE}.

It is worth noting that [28] also uses two-wayness to simulate one-counter automata, but in a game-based setting (the latter is reflected by alternation).

Let us show how to build an A2A from an OCAP and a target state.

**Lemma 7.** Let \( A = (Q, \Sigma, q_0, \Delta) \) be an OCAP and \( q_f \in Q \). There is an A2A \( \mathcal{T} = (S, \Sigma, s_m, \delta, S_f) \), with \( \Sigma = \Sigma \cup \{ \square \} \), such that \( L(\mathcal{T}) = \{ w \in \mathcal{W}_\Sigma \mid q_f \text{ is } \gamma_w\text{-reachable} \} \). Moreover, \(|\mathcal{T}| = O(|A|^2)\).

\(^1\) One often considers a transition function \( \delta : S \times \Sigma \times \{ \text{first}^?, \neg \text{first}^? \} \rightarrow B^+(S \times \{ +1, 0, -1 \}) \), but a relation is more convenient for us.
The Complexity of Flat Freeze LTL

Proof. The states of $\mathcal{T}$ include $Q$ (to simulate $\mathcal{A}$), a new initial state $s_0$, and some extra states, which will be introduced below.

Starting in $s_0$ and at the first letter of the input word, the A2A $\mathcal{T}$ spawns several copies: $s_0 \xrightarrow{\mathcal{T}} (q_0, 0) \land \bigwedge_{x \in \mathcal{X}} (\text{search}(x), +1)$. The copy starting in $q_0$ will henceforth simulate $\mathcal{A}$. Moreover, from the new state $\text{search}(x)$, we will check that $x$ occurs exactly once in the input word. This is accomplished using the transitions $\text{search}(x) \xrightarrow{\mathcal{T}} (\vee_x, +1)$, as well as $\text{search}(x) \xrightarrow{\mathcal{T}} (\text{search}(x), +1)$ and $\vee_x \xrightarrow{\mathcal{T}} (\vee_x, +1)$ for all $y \in \Sigma \setminus \{x\}$. Thus, $\vee_x$ signifies that $x$ has been seen and must not be encountered again. Since the whole word has to be scanned, $\vee_x$ is visited infinitely often so that we set $\vee_x \in S_f$.

It remains to specify how $\mathcal{T}$ simulates $\mathcal{A}$. The underlying idea is very simple. A configuration $(q, v) \in \mathcal{C}_\mathcal{A}$ of $\mathcal{A}$ corresponds to the configuration/proof obligation $(q, i)$ of $\mathcal{T}$ where position $i$ is the $(v+1)$-th occurrence of $\square$ in the input parameter word. The simulation then proceeds as follows.

Increment/decrement: To mimic a transition $(q, +1, q') \in \Delta$, the A2A $\mathcal{T}$ simply goes to the next occurrence of $\square$ on the right hand side and enters $q'$. This is accomplished by several A2A-transitions: $q \xrightarrow{\mathcal{T}} (\text{right}(q'), +1)$, right$(q') \xrightarrow{\mathcal{T}} (\text{right}(q'), +1)$ for every $x \in \mathcal{X}$, and right$(q') \xrightarrow{\mathcal{T}} (q', 0)$. Decrement is handled similarly, using states of the form left$(q')$.

Finally, $(q, 0, q') \in \Delta$ is translated to $q \xrightarrow{\mathcal{T}} (q', 0)$.

Equality test: A zero test $(q, \text{zero}^?, q') \in \Delta$ corresponds to $q \xrightarrow{\text{first}^?} (q', 0)$. To simulate a transition $(q, =x, q') \in \Delta$, the A2A spawns two copies. One goes from $q$ to $q'$ and stays at the current position. The other goes to the right and accepts if it sees $x$ before hitting another $\square$-symbol. Thus, we introduce a new state present$(x)$ and transitions $q \xrightarrow{\mathcal{T}} (q', 0) \land \text{present}(x)$, $\text{present}(x) \xrightarrow{\mathcal{T}} \text{true}$, and $\text{present}(x) \xrightarrow{\mathcal{T}} (\text{present}(x), +1)$ for all $y \in \mathcal{X} \setminus \{x\}$. Note that there is no transition that allows $\mathcal{T}$ to read $\square$ in state present$(x)$.

Inequality tests: To simulate a test of the form $x > w$, we proceed similarly to the previous case.

The A2A generates a branch in charge of verifying that the parameter $x$ lies strictly to the left of the current position. Note that transitions with label $<x$ are slightly more subtle, as $x$ has to be retrieved strictly beyond the next delimiter $\square$ to the right of the current position. More precisely, $(q, <x, q') \in \Delta$ translates to $q \xrightarrow{\mathcal{T}} (q', 0) \land \text{search}^+(x)$, $\text{search}^+(x)$ until we encounter the next occurrence of $\square$. We then go into $\text{search}(x)$ (introduced at the beginning of the proof), which will be looking for an occurrence of $x$. Formally, we introduce $\text{search}^+(x) \xrightarrow{\mathcal{T}} (\text{search}^+(x), +1)$ for all $y \in \mathcal{X} \setminus \{x\}$, and $\text{search}^+(x) \xrightarrow{\mathcal{T}} (\text{search}(x), +1)$.

In state $q_f$, we accept: $q_f \xrightarrow{\square} \text{true}$. The only infinite branches in an accepting run are those that eventually stay in a state of the form $\vee_x$. Thus, we set $S_f = \{\vee_x \mid x \in \mathcal{X}\}$. It is not hard to see that $L(T) = \{w \in W_X \mid q_f \text{ is } \gamma_w\text{-reachable}\}$. Note that $T$ has a linear number of states and a quadratic number of transitions (some transitions of $\mathcal{A}$ are simulated by $O(|\mathcal{X}|)$-many transitions of $\mathcal{T}$).

A similar reduction takes care of Büchi reachability. Together with Theorem 6, we thus obtain the following:

▶ Corollary 8. OCAP-REACHABILITY and OCAP-BÜCHI are both in PSPACE.

Note that we are using alternation only to a limited extent. We could also reduce reachability in OCAPs to non-emptiness of the intersection of several two-way automata. However, this would require a more complicated word encoding of parameter instantiations, which then have to include letters of the form $<x$ and $x>$. 


3.2 OCAP-Reachability is in NP

We will now show that we can improve the upper bounds given in Corollary 8 to \(NP\), which is then optimal: \(NP\)-hardness of both problems, even in the presence of a single parameter, can be proved using a straightforward reduction from the non-emptiness problem for nondeterministic two-way automata over finite words over a unary alphabet, which is \(NP\)-complete [17].

In a first step, we exploit our reduction from OCAPs to A2As to establish a bound on the parameter values. To solve the reachability problems, it will then be sufficient to consider parameter instantiations up to that bound. For the rest of this section, unless stated otherwise, we fix an OCAP \(A = (Q, \mathcal{X}, q_0, \Delta)\) and a state \(q_f \in Q\).

\[\blacktriangleright\text{Lemma 9.}\] There is \(d \in 2^{O(|A|^4)}\) such that the following holds: If \(A \models \text{Reach}(q_f)\), then there is a parameter instantiation \(\gamma : \mathcal{X} \rightarrow \mathbb{N}\) such that \(\gamma(x) \leq d\) for all \(x \in \mathcal{X}\) and \(q_f\) is \(\gamma\)-reachable.

\[\blacktriangleright\text{Proof.}\] According to Lemma 7, there is an A2A \(T = (S, \mathcal{X} \cup \{\square\}, s_0, \delta, S_f)\) such that \(L(T) = \{w \in W_{\mathcal{X}} \mid q_f\text{ is }\gamma_w\text{-reachable}\}\) and \(|T| = O(|A|^2)\). By [30], there is a (nondeterministic) Büchi automaton \(B\) such that \(L(B) = L(T)\) and \(|B| \in 2^{O(|T|^2)}\) (cf. also [8]).

Let \(d = |B|\). Suppose \(A \models \text{Reach}(q_f)\). This implies that \(L(B) \neq \emptyset\). But then, there must be a word \(u \in \Sigma^*\) such that \(|u| \leq |B|\) (\(|u|\) denoting the length of \(u\)) and \(w = u\square^i \in L(B)\). We have that \(q_f\) is \(\gamma_w\)-reachable and \(\gamma_w(x) \leq |B| = d\) for all \(x \in \mathcal{X}\).

As a corollary, we obtain that it is sufficient to consider bounded runs only. For a parameter instantiation \(\gamma : \mathcal{X} \rightarrow \mathbb{N}\) and a bound \(d \in \mathbb{N}\), we say that a \(\gamma\)-run \(\rho = (q_0, v_0) \rightarrow_{\gamma} (q_1, v_1) \rightarrow_{\gamma} (q_2, v_2) \rightarrow_{\gamma} \cdots\) is \(d\)-bounded if all its counter values \(v_0, v_1, \ldots\) are at most \(d\).

\[\blacktriangleright\text{Corollary 10.}\] There is \(d \in 2^{O(|A|^4)}\) such that the following holds: If \(A \models \text{Reach}(q_f)\), then there is a parameter instantiation \(\gamma : \mathcal{X} \rightarrow \mathbb{N}\) such that \(\gamma(x) \leq d\) for all \(x \in \mathcal{X}\) and \(q_f\) is reachable within a \(d\)-bounded \(\gamma\)-run.

\[\blacktriangleright\text{Proof.}\] Consider \(d \in 2^{O(|A|^4)}\) due to Lemma 9. A standard argument in OCAs with \(n\) states is that, for reachability, it is actually sufficient to consider runs up to some counter value in \(O(n^3)\) [25]. We can apply the same argument here to deduce, together with Lemma 9, that parameter and counter values can be bounded by \(d + O(|Q|^3)\).

The algorithm that solves OCAP-REACHABILITY in \(NP\) will guess a parameter instantiation \(\gamma\) and a maximal counter value in binary representation (thus, of polynomial size). It then remains to determine, in polynomial time, whether the target state is reachable under this guess. To this end, we will exploit the following lemma due to Galil [17].

Let \(A = (Q, q_0, \Delta)\) be an OCA, \(q, q' \in Q\), and \(v, v' \in \mathbb{N}\). A run \((q_0, v_0) \rightarrow (q_1, v_1) \rightarrow \cdots \rightarrow (q_{n-1}, v_{n-1}) \rightarrow (q_n, v_n)\) of \(A\), \(n \in \mathbb{N}\), is called a \((v, v')\)-run if \(\min\{v, v'\} < v_i < \max\{v, v'\}\) for all \(i \in \{1, \ldots, n - 1\}\). In other words, all intermediate configurations have counter values strictly between \(v\) and \(v'\).

\[\blacktriangleright\text{Lemma 11}\] ([17]). The following problems can be solved in polynomial time:

**Input:** An OCA \(A = (Q, q_0, \Delta)\), \(q, q' \in Q\), and \(v, v' \in \mathbb{N}\) given in binary.

**Question 1:** Is there a \((v, v')\)-run from \((q, v)\) to \((q', v')\)?

**Question 2:** Is there a \((v, v')\)-run from \((q, v)\) to \((q', v)\)?

Actually, [17] uses the polynomial-time algorithms stated in Lemma 11 to show that non-emptiness of two-way automata on finite words over a unary alphabet is in \(NP\) (cf. the
proof of the corresponding lemma on page 84 of [17], where the endmarkers of a given word correspond to \(v\) and \(v'\).

We can now deduce the following result, which is an important step towards model checking, but also of independent interest.

\begin{theorem}
The Complexity of Flat Freeze LTL
\end{theorem}

\begin{proof}
Let \(A = (Q, \mathcal{X}, q_f, \Delta)\) be an OCAP with \(\mathcal{X} = \{x_1, \ldots, x_n\}\), and let \(q_f \in Q\) be a state of \(A\). Without loss of generality, we consider reachability of the configuration \((q_f, 0)\) (since one can add a new target state and a looping decrement transition).

Our nondeterministic polynomial-time algorithm proceeds as follows. First, it guesses \(d \in 2^{O(|\mathcal{A}|^4)}\) depending on the size with respect to \(|\mathcal{A}|\), as well as some parameter instantiation \(\gamma\) satisfying \(\gamma(x_i) \leq d\) for all \(1 \leq i \leq n\) where each \(d_i := \gamma(x_i)\) is represented in binary, too. We may assume \(n \geq 1\) and that \(d_0 < d_1 < d_2 < \cdots < d_n < d_{n+1}\), where \(d_0 = 0\) and \(d_{n+1} = d\) (otherwise, we can rename the parameters accordingly). Second, the algorithm checks that there exists a \(d\)-bounded \(\gamma\)-run of the form \((q_0,v_0) \rightarrow^* (q'_0,v_0) \rightarrow^* (q_1,v_1) \rightarrow^* (q'_1,v_1) \rightarrow^* \cdots \rightarrow^* (q_k,v_k) \rightarrow^* (q'_k,v_k)\) such that (letting \(D = \{d_0,d_1,\ldots,d_n,d_{n+1}\}\))

1. \((q_0,v_0) = (q_f,0)\) and \((q_k,v_k) = (q_f,0)\),
2. \(v_j \in D\) for all \(j \in \{0,\ldots,k\}\),
3. between \((q_j,v_j)\) and \((q'_j,v_j)\), the counter values always equal \(v_j\), for all \(j \in \{0,\ldots,k\}\),
4. (strictly) between \((q'_j,v_j)\) and \((q_{j+1},v_{j+1})\), the counter values are always different from the values in \(D\), for all \(j \in \{0,\ldots,k-1\}\).

Note that we can assume \(k \leq |Q| \cdot (|\mathcal{X}| + 2)\), since in every longer run, the exact same configuration is encountered at least twice.

Hence, to check whether there exists a finite initialized \(d\)-bounded \(\gamma\)-run \((q_0,v_0) \rightarrow^* (q_f,0)\), it is sufficient to guess \(2k\) configurations, where \(k \leq |Q| \cdot (|\mathcal{X}| + 2)\), and to verify that these configurations contribute to constructing a run of the form described above. This can be checked in polynomial time: The case (3) is obvious, and (4) is due to Lemma 11.

\end{proof}

\section{From OCAP-Reachability to SOCA-Model-Checking}

This section is dedicated to model checking OCAs against flatLTL\(^4\). In Section 4.1, we establish a couple of reductions from \textsc{SOCA-MC}(flatLTL\(^4\)) down to \textsc{OCAP-Reachability}, which allow us to conclude that the former problem is in \textsc{NEXPTIME}. Then, in Section 4.2, we provide a matching lower bound.

\subsection{Upper Bounds}

First, we use the fact that \textsc{OCAP-Reachability} is in \textsc{NP} (Theorem 12) to show that \textsc{OCAP-Büchi} is in \textsc{NP}, too:

\begin{lemma}
The Complexity of Flat Freeze LTL
\end{lemma}

\begin{proof}
Let \(A = (Q, \mathcal{X}, q_f, \Delta)\) be an OCAP with \(\mathcal{X} = \{x_1, \ldots, x_n\}\) (without loss of generality, we suppose \(n \geq 1\)), and let \(F \subseteq Q\). It is sufficient to give an algorithm that determines whether a particular state \(q_f \in F\) can be reached infinitely often. Thus, we construct an OCAP \(A'\), with a distinguished state \(s\), such that \(A \models \text{Reach}^\omega(q_f)\) iff \(A' \models \text{Reach}(s)\).

To this end, we first define an OCA \(M\), which is obtained from \(A\) by (i) removing all transitions whose labels are of the form \(\text{zero}\), \(\equiv x\), or \(< x\), for some \(x \in \mathcal{X}\), and (ii) replacing all transition labels \(\equiv x\) (for some \(x \in \mathcal{X}\)) by \(0\). Then, we compute the set \(T \subseteq Q\) of states
Figure 2 The OCAP $\mathcal{A}'$ constructed from $\mathcal{A}$ in the proof of Lemma 13.

$t$ of $\mathcal{M}$ from which there is an infinite run starting in $(t, 0)$ and visiting $q_f$ infinitely often. This can be done in NLOGSPACE [9].

Now, we obtain the desired OCAP $\mathcal{A}' = (Q', \mathcal{X}', q_{in}, \Delta')$ as follows. The set of states is $Q' = Q \cup \{q \mid q \in Q\} \cup \{s, s, t_1, \ldots, t_n\}$. That is, we introduce a copy $\hat{q}$ for every $q \in Q$ as well as fresh states $s, \hat{s}, t_1, \ldots, t_n$. Recall that $\hat{s}$ will be the target state of the reachability problem we reduce to. Moreover, $\mathcal{X}' = \mathcal{X} \cup \{y\}$ where $y$ is a fresh parameter. It remains to define the transition relation $\Delta'$, which includes $\Delta$ as well as some new transitions (cf. Figure 2).

Essentially, there are two cases to consider:

1. First, $q_f$ may be visited infinitely often in $\mathcal{A}$, and infinitely often along with the same counter value. To take this case into account, we use the new parameter $y$ and a new transition $(q_f, y, s)$, which allows $\mathcal{A}'$ to “store” the current counter value in $y$. From $s$, we then enter and simulate the copy of $\mathcal{A}$, i.e., we introduce transitions $(s, \text{op}, \hat{q})$ for all $(q_f, \text{op}, q) \in \Delta$, as well as $(\hat{q}_1, \text{op}, \hat{q}_2)$ for all $(q_1, \text{op}, q_2) \in \Delta$. If, in the copy, $\hat{q}_f$ is visited along with the value stored in $y$, we may thus enter $\hat{s}$.

2. Suppose, on the other hand, that a $\gamma$-run $\rho$ of $\mathcal{A}$ visits $q_f$ infinitely often, but not infinitely often with the same counter value. If $\rho$ visits some counter value $v \leq \max\{\gamma(x) \mid x \in \mathcal{X}\}$ infinitely often, then it necessarily contains a subrun of the form $(q, v) \xrightarrow{\gamma} (q_f, v') \xrightarrow{\gamma} (q, v)$ for some $q \in Q$ and $v' \in \mathbb{N}$. But then, there is an infinite $\gamma$-run that visits $(q_f, v')$ infinitely often so that case (1) above applies. Otherwise, $\rho$ will eventually stay strictly above $\max\{\gamma(x) \mid x \in \mathcal{X}\}$. Thus, we add the following transitions to $\Delta'$, which will allow $\mathcal{A}'$ to move from $t \in T$ to $\hat{s}$ provided the counter value is greater than the maximal parameter value: $(t, 0, t_1)$ for all $t \in T$, as well as $(t_1, >x_1, t_2), (t_2, >x_2, t_3), \ldots, (t_n, >x_n, \hat{s})$.

From Theorem 12 and Lemma 13, we obtain the exact complexity of OCAP-Büchi (the lower bound is by a straightforward reduction from OCAP-Reachability).

**Corollary 14.** OCAP-Büchi is NP-complete.

The link between OCA-MC(\text{flatLTL}$^+$) and OCAP-Büchi is due to [13]:

**Lemma 15 ([13]).** Let $\mathcal{A}$ be an OCA and $\varphi \in$ flatLTL$^+$ be a sentence. One can compute, in exponential time, an OCAP $\mathcal{A}' = (Q, \mathcal{X}, q_{in}, \Delta)$ (of exponential size) and a set $F \subseteq Q$ such that $\mathcal{A} \models \varphi$ iff $\mathcal{A}' \models \text{Reach}^2(q_f)$ for some $q_f \in F$.

Actually, the construction from [13] can be easily extended to handle register tests of the form $<r$ and $>r$. However, the restriction to flatLTL$^+$ is crucial. It allows one to assume that a register is written at most once so that, in the OCAP, it can be represented as a parameter. In particular, $\mathcal{X}$ is the set of registers that occur in $\varphi$. From Corollary 14 and Lemma 15, we now deduce our first model-checking result:

**Corollary 16.** OCA-MC(\text{flatLTL}$^+$) is in NEXPTIME.
The reason is that succinct updates can be encoded in small LTL formulas. The new propositions are depicted in dashed boxes. States all second-last delimiters. Finally, we write last encoding. For example, bits (z), represent the bits needed to encode z in binary. At the transitions originating from n and n’, the counter is updated by +1 or −1, depending on whether z is positive or negative, respectively.

Proof. Let A = (Q, q0, Δ, μ) be a SOCA and ϕ ∈ flatLTL be a sentence. Without loss of generality, we assume that ϕ is in negation normal form, i.e., negation is applied only to atomic propositions. This is thanks to the logical equivalences ¬(ϕ1 U ϕ2) ≡ ¬ϕ1 R ¬ϕ2 and ¬(ϕ1 R ϕ2) ≡ ¬ϕ1 U ¬ϕ2. We construct an OCA A’ = (Q’, q0’, Δ’, μ’) and a sentence ϕ’ ∈ flatLTL of polynomial size such that A |= ϕ if A’ |= ϕ’.

Let Z = {z | (q, z, q’) ∈ Δ ∩ (Q × Z × Q) with |z| ≥ 2} and let Λ = {#z | z ∈ Z} ∪ {0, 1} be a set of fresh propositions. To obtain A’ from A, we replace every transition (q, z, q’) ∈ Δ ∩ (Q × Z × Q) by the gadget depicted in Figure 3. The gadget implements a binary counter generating sequences (of sets of propositions) of the form depicted in Figure 4, for z = 6. We assume that the least significant bit is on the left. To make sure that the binary counter works as requested, we define an LTL formula Counter.

Towards its definition, let us first introduce some notation and abbreviations. For z ∈ Z, let bits(z) denote the number of bits needed to represent the binary encoding of |z|. For example, bits(6) = 3. For i ∈ {1, ..., bits(z)}, we let biti(z) denote the i-th bit in that encoding. For example, (bit1(6), bit2(6), bit3(6)) = (0, 1, 1). In the following, we write Λ for \( \bigvee_{\gamma \in \Lambda} \neg \gamma \) and \( \neg \Lambda \) for \( \bigwedge_{\gamma \in \Lambda} \neg \gamma \). For an illustration of the following LTL formulas, we refer to Figure 4. Formula first(#z) = ¬Λ U X¬#z holds right before a first delimiter symbol. Similarly, last(#z) = #z U X¬#z identifies all last delimiters and last¬(#z) = #z U ((0 V 1) U last(#z)) all second-last delimiters. Finally, we write r(ψ) for Xbits(#z)+1 ψ.

Now, Counter = Init \& Fin \& Inc \& Exit is the conjunction of the following formulas:

- **Init** says that the first binary number is always one:

Init = \( \bigwedge_{z \in Z} G(\text{first(#z)} \rightarrow X^2(1 \& X(0 U \#z))) \)
Their proof can be adapted in a straightforward way: In Figure 3 of [18], we replace the non-emptiness of its product with the given OCAP using Corollary 14. This subsection presents a matching lower bound for model checking flatLTL formulas from [18], where Göller et al. show the NEXPTIME-hardness of LTL model checking OCA with parametric and succinct updates [18].

Proof. NEXPTIME-hardness of SOCA is due to [18], where Göller et al. show NEXPTIME-hardness of LTL model checking OCA with parametric and succinct updates [18]. Their proof can be adapted in a straightforward way: In Figure 3 of [18], we replace the parametric update +x by an equality test =x and add a self-loop to the start state with label +1. We remark that this lower bound already holds for SOCA that use only one parameter x and all parameterized tests are of the form =x.

Moreover, SOCA can be reduced to OCA in polynomial time, using the construction from Lemma 17.

Membership of OCA in NEXPTIME is by a standard argument. The given LTL formula can be translated into a Büchi automaton of exponential size. Then, we check non-emptiness of its product with the given OCA using Corollary 14.

\[ Fin = \bigwedge_{z \in Z} G(\text{last}^{-}(\#z) \rightarrow \bigwedge_{i=1}^{\text{bits}(z)} X^i \text{bit}_i(z)) \]

Inc implements the increments:

\[ Inc = \bigwedge_{z \in Z} G\left( \rightarrow X((1 \land \gamma_{\#z} 0) \cup (0 \land \gamma_{\#z} 1 \land X \text{suff}_{\#z}^z) ) \right) \]

where \( \text{suff}_{\#z}^z = ((0 \land \gamma_{\#z} 0) \lor (1 \land \gamma_{\#z} 1)) \cup \#z \) verifies that the suffixes (w.r.t. the current position) of the current and the following binary number coincide.

\[ Exit = \bigwedge_{z \in Z} G(\text{first}(\#z) \rightarrow F \text{last}(\#z)) \]

Note that the gadget makes sure that, in between two delimiters \#z, there are exactly \( \text{bits}(z) \)-many bits.

Now, we set \( \varphi' = [\varphi] \land \text{Counter} \). Here, \([\varphi]\) simulates \( \varphi \) on all positions that do not contain propositions from \( \Lambda \). It is inductively defined as follows:

- \([p] = p\)
- \([-p] = \neg p\)
- \([0r] = 0r\)
- \([1r] = 1r]\)
- \([\psi_1 \lor \psi_2] = [\psi_1] \lor [\psi_2]\)
- \([\psi_1 \land \psi_2] = [\psi_1] \land [\psi_2]\)
- \([\neg \psi] = \neg [\psi]\)
- \([\psi \land \varphi] = \psi \land [\varphi]\)
- \([\psi \lor \varphi] = \psi \lor [\varphi]\)
- \([\psi \land \varphi] = \psi \land [\varphi]\)
- \([\psi \lor \varphi] = \psi \lor [\varphi]\)
- \([\psi R \psi_1] = (\neg \psi \lor [\psi_1]) R (\neg \psi \land [\psi_1])\)

Note that, since \( \varphi \) was required to be in negation normal form, \( \varphi' \) is indeed in flatLTL$^\downarrow$. ▶

Corollary 16 and Lemma 17 imply the NEXPTIME upper bound for SOCA model checking:

\[ \text{corollary18} \]

\subsection{Lower Bounds}

This subsection presents a matching lower bound for model checking flatLTL$^\downarrow$. We first state NEXPTIME-hardness of OCA-MC(LTL), which we reduce, in a second step, to OCA-MC(flatLTL$^\downarrow$).

\[ \text{theorem19} \]

\[ \text{corollary14} \]
The Complexity of Flat Freeze LTL

Lemma 20. OCA-MC(flattLTL) and SOCA-MC(flattLTL) are NEXPTIME-hard.

Proof. We present a reduction from OCAP-MC(LTL), which is NEXPTIME-complete by Theorem 19. Let \( A = (Q, X, q_0, \Delta, \mu) \) be an OCAP and let \( \varphi \) be an LTL formula. We define an OCA \( A' = (Q', q'_0, \Delta', \mu') \) and a sentence \( \varphi' \in \text{flattLTL}^2 \) such that \( A \models \varphi \) iff \( A' \models \varphi' \).

Without loss of generality, by [18] (cf. proof of Theorem 19), we may assume that \( A \) uses only one parameter \( x \) and that all parameterized tests are of the form \( =x \).

The idea is as follows. A new initial state \( q'_0 \), in which only a fresh proposition \( \text{freeze} \) holds, will allow \( A' \) to go to an arbitrary counter value, say, \( v \). The flattLTL\(^2\) formula of the form \( F(\text{freeze} \land \downarrow_r \psi) \) stores \( v \), which will henceforth be interpreted as parameter \( x \). Hereby, formula \( \psi \) makes sure that, whenever \( A \) performs an equality check \( =x \), the current counter value coincides with \( v \). To do so, we create two copies of the original state space: \( Q \times \{0, 1\} \).

A transition \((q, =x, q')\) of \( A \) is then simulated, in \( A' \), by a 0-labeled transition to \((q', 1)\). States of the latter form are equipped with a fresh proposition \( p_{=x} \) indicating that the current counter value has to coincide with the contents of \( r \). Thus, we can set \( \psi = G(p_{=x} \rightarrow =r) \).

Formally, \( A' \) is given as follows. We let \( Q' = (Q \times \{0, 1\}) \cup \{q'_0\} \) and

\[
\Delta' = \{(q'_0, \text{op}, q_0) | \text{op} \in \{+1, -1\}\} \cup \{(q'_0, \text{zero}', (q_0, 0))\}
\]

\[
\cup \{(q, b, 0, (q, 1)) | (q, =x, q') \in \Delta \text{ and } b \in \{0, 1\}\}
\]

\[
\cup \{(q, b, \text{op}, (q, 0)) | (q, \text{op}, q') \in \Delta \setminus (Q \times \{=x\} \times Q) \text{ and } b \in \{0, 1\}\}.
\]

Moreover, we set \( \mu'(q'_0) = \{\text{freeze}\} \) as well as \( \mu'((q, 0)) = \mu(q) \) and \( \mu'((q, 1)) = \mu(q) \cup \{p_{=x}\} \) for all \( q \in Q \). Finally, \( \varphi' = F(\text{freeze} \land \downarrow_r G(p_{=x} \rightarrow =r)) \land (\text{freeze} \cup \neg \text{freeze} \land \varphi) \). Note that the subformula \( \text{freeze} \cup \neg \text{freeze} \land \varphi \) makes sure that \( \varphi \) is satisfied as soon as we enter the original initial state \( q_0 \) (actually, \((q_0, 0)\)).

We conclude stating the exact complexity of model checking (S)OCAs against flattLTL\(^2\):

Theorem 21. OCA-MC(flattLTL\(^2\)) and SOCA-MC(flattLTL\(^2\)) are NEXPTIME-complete.

5 Conclusion

In this paper, we established the precise complexity of model checking OCAs and SOCA against flat freeze LTL. To do so, we established a tight link between OCAPs and alternating two-way automata over words. Exploiting alternation further, it is likely that we can obtain positive results for one-counter games with parameterized tests. Another interesting issue for future work concerns model checking OCAPs, which, in this paper, is defined as the following question: Are there a parameter instantiation \( \gamma \) and a \( \gamma \)-run that satisfies the given formula? In fact, it also makes perfect sense to ask whether there is such a run for all parameter instantiations, in particular when requiring that all system runs satisfy a given property. We believe that our techniques can be used also in that case.

Although OCAPs have mainly been used for deciding a model-checking problem for OCAs, they constitute a versatile tool as well as a natural stand-alone model. In fact, parameterized extensions of a variety of system models have been explored recently, among them OCAs with parameterized updates [22, 19], and parametric timed automata [2, 5] where clocks can be compared with parameters (similarly to parameterized tests in OCAPs). Our results may have an impact on those models, too, thus beyond their application to model checking flat freeze LTL of OCAs. The (existential) non-emptiness problem for parametric timed automata with a single parametric clock (IPTAs) was first studied and solved by Alur, Henzinger, and Vardi, who provided a nonelementary decision procedure [2]. Recently, Bundala and
Ouaknine improved this significantly to $2\text{NEXPTIME}$ [5]. They provide a polynomial-time reduction to the reachability problem for OCAPs where, in addition to parameters, the counter values can be compared with constants $c \in \mathbb{N}$ encoded in binary. Altogether, they obtain a $2\text{NEXPTIME}$ algorithm for the non-emptiness problem for 1PTAs. Therefore, it will be worthwhile to study whether our NP upper bound for reachability in OCAPs applies to extended OCAPs, too (however, with binary updates, the problem is already PSPACE-hard [14]). This would yield an optimal NEXPTIME algorithm for non-emptiness in 1PTAs.

References

The Complexity of Flat Freeze LTL


Higher-Order Linearisability

Andrzej S. Murawski\(^1\) and Nikos Tzevelekos\(^2\)

1 University of Warwick, Coventry, UK
2 Queen Mary University of London, London, UK

Abstract

Linearisability is a central notion for verifying concurrent libraries: a library is proven correct if its operational history can be rearranged into a sequential one that satisfies a given specification. Until now, linearisability has been examined for libraries in which method arguments and method results were of ground type. In this paper we extend linearisability to the general higher-order setting, where methods of arbitrary type can be passed as arguments and returned as values, and establish its soundness.

1998 ACM Subject Classification F.1.2 Models of Computation

Keywords and phrases Linearisability, Concurrency, Higher-Order Computation

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2017.34

1 Introduction

Software libraries provide implementations of routines, often of specialised nature, to facilitate code reuse and modularity. To support the latter, they should follow specifications that describe the range of acceptable behaviours for correct and safe deployment. Adherence to specifications can be formalised using the classic notion of contextual approximation (refinement), which scrutinises the behaviour of code in any possible context. Unfortunately, the quantification makes it difficult to prove contextual approximations directly, which motivates research into sound techniques for establishing it.

In the concurrent setting, a notion that has been particularly influential is that of linearisability [12]. Linearisability requires that, for each history generated by a library, one should be able to find another history from the specification (a linearisation), which matches the former up to certain rearrangements of events. In the original formulation by Herlihy and Wing [12], these permutations were not allowed to disturb the order between library returns and client calls. Moreover, linearisations were required to be sequential traces, that is, sequences of method calls immediately followed by their returns.

In this paper we shall work with open higher-order libraries, which provide implementations of public methods and may themselves depend on abstract ones, to be supplied by parameter libraries. The classic notion of linearisability only applies to closed libraries (without abstract methods). Additionally, both method arguments and results had to be of ground type. The closedness limitation was recently lifted in [13, 3], which distinguished between public (or implemented) and abstract methods (callable). Although [13] did not in principle exclude higher-order functions, those works focussed on linearisability for the case where the allowable methods were restricted to first-order functions (int → int). Herein, we give a systematic exposition of linearisability for general higher-order concurrent libraries,

\(^*\) Research was partially supported by the EPSRC (EP/P004172/1).
where methods can be of arbitrary higher-order types. In doing so, we also propose a corresponding notion of sequential history for higher-order library interactions.

We examine libraries \( L \) that can interact with their environments by means of public and abstract methods: a library \( L \) with abstract methods of types \( \Theta = \theta_1, \ldots, \theta_n \) and public methods \( \Theta' = \theta'_1, \ldots, \theta'_{n'} \) is written as \( L : \Theta \rightarrow \Theta' \). We shall work with arbitrary higher-order types generated from the ground types unit and int. Types in \( \Theta, \Theta' \) must always be function types, i.e. their order is at least 1.

A library \( L \) may be used in computations by placing it in a context that will keep on calling its public methods (via a client \( K \)) as well as providing implementations for the abstract ones (via a parameter library \( L' \)). The setting is depicted in Figure 1. Note that, as the library \( L \) interacts with \( K \) and \( L' \), they exchange functions between each other. Consequently, in addition to \( K \) making calls to public methods of \( L \) and \( L \) making calls to its abstract methods, \( K \) and \( L' \) may also issue calls to functions that were passed to them as arguments during higher-order interactions. Analogously, \( L \) may call functions that were communicated to it via library calls.

Our framework is operational in flavour and draws upon concurrent [15, 7] and operational game semantics [14, 16, 8]. We shall model library use as a game between two participants: Player \( (P) \), corresponding to the library \( L \), and Opponent \( (O) \), representing the environment \((L', K)\) in which the library was deployed. Each call will be of the form \( \text{call} \ m(v) \) with the corresponding return of the shape \( \text{ret} \ m(v) \), where \( v \) is a value. As we work in a higher-order framework, \( v \) may contain functions, which can participate in subsequent calls and returns. Histories will be sequences of moves, which are calls and returns paired with thread identifiers. A history is sequential just if every move produced by \( O \) is immediately followed by a move by \( P \) in the same thread. In other words, the library immediately responds to each call or return delivered by the environment. In contrast to classic linearisability, the move by \( O \) and its response by \( P \) need not be a call/return pair, as the higher-order setting provides more possibilities (in particular, the \( P \) response may well be a call). Accordingly, linearisable higher-order histories can be seen as sequences of atomic segments (linearisation points), starting at environment moves and ending with corresponding library moves.

In the spirit of [3], we are going to consider two scenarios: one in which \( K \) and \( L' \) share an explicit communication channel (the general case) as well as a situation in which they can only communicate through the library (the encapsulated case). Further, we also handle the case in which extra closure assumptions can be made about the parameter library (the relational case), which can be useful for dealing with a variety of assumptions on the use of parameter libraries that may arise in practice. In each case, we present a candidate definition of linearisability and illustrate it with tailored examples. The suitability of each kind of linearisability is demonstrated by showing that it implies the relevant form of contextual approximation (refinement). We also examine compositionality of the proposed concepts.
One of our examples will discuss the implementation of the flat-combining approach [11, 3], adapted to higher-order types.

1.1 Example: a higher-order multiset library

Higher-order libraries are common in languages like ML, Java, Python, etc. As an illustrative example, we consider a library written in ML-like syntax which implements a multiset data structure with integer elements. For simplicity, we assume that its signature contains just two methods:

\[
\text{count} : \text{int} \rightarrow \text{int}, \quad \text{update} : (\text{int} \times (\text{int} \rightarrow \text{int})) \rightarrow \text{int}.
\]

The former method returns for each integer its multiplicity in the multiset – this is 0 if the integer is not a member of the multiset. On the other hand, update takes as an argument an integer \(i\) and a function \(f\), and updates the multiplicity of \(i\) in the multiset to \(|f(i)|\) (we use the absolute value of \(f(i)\) in order to meet the multiset requirement that element multiplicities not be negative; alternatively, we could have used exceptions to quarantine such client method behaviour). Methods with the same functionalities can be found e.g. in the multiset module of the ocaml-containers library [1]. While our example is simple, the same kind of analysis as below can be applied to more intricate examples such as map methods for integer-valued arrays, maps or multisets.

Example 1 (Multiset). Consider the concurrent multiset library \(L_{\text{ms}}\) in Figure 2. It uses a private reference for storing the multiset’s characteristic function and reads optimistically, without locking (cf. [10, 19]). The update method in particular reads the current multiplicity of the given element \(i\) (via count) and computes its new multiplicity without acquiring a lock on the characteristic function. It only acquires a lock when it is ready to write the new value (line 10) in the hope that the value at \(i\) will still be the same and the update can proceed; if not, another attempt to update the value is made.

Let us look at some example executions of the library via their resulting histories, i.e. sequences of method calls and returns between the library and a client. In the topmost block (a) of history diagrams of Figure 2, we see three such executions. Note that we do not record internal calls to count or aux, and use \(m\) and variants for method identifiers (names). We use the abbreviation \(cnt\) for count, and \(upd\) for update, and initially ignore the circled events for \(cnt\). Each execution involves 2 threads.

In the first execution, the client calls update\((i, m)\) in the second thread, and subsequently calls count\((i)\) in the first thread. The code for update stipulates that first count\((i)\) be called internally, returning some multiplicity \(j\) for \(i\), and then \(m(j)\) should be called. As soon \(m\) returns a value \(j'\), update sets the multiplicity of \(i\) to \(j'\) and itself returns \(j'\). The last event in this history is a return of count in the first thread with the old value \(j\). According to our proposed definition, this history will be linearisable to another, intuitively correct one: the last return can be moved to the circled position. At this point the notion of linearisability is used informally, but it will be made precise in the following sections. In the second execution, the last return of count in the first thread returns the updated value. In this case, we will be able to move call count\((i)\) to the circled position to obtain a linearisation, which is obviously correct. Finally, in the third execution we have a history that will turn out non-linearisable to an intuitively correct history. Indeed, we should not be able to return the updated value in the first thread before \(m\) has returned it in the second one.

The two histories in block (b) in the same figure demonstrate the mechanism for updates. The first history will be linearisable to the second one. In the second history we see that both
threads try to update the same element \( i \), but the first one succeeds in it first and returns \( k \) on \( \text{update} \). Then, the second thread realises that the value of \( i \) has been updated to \( k \) and calls \( m \) again, this time with argument \( k \). An important feature of the second history is that it is sequential: each client event (call or return) is immediately followed by a library event.

Observe that the rearrangements discussed above involve either advancing a library action or postponing an environment action and that each action could be a call or a return. Definition 6 will capture this formally. For now, we note that this generalises the classic setting [12], where library method returns could be advanced and environment calls deferred.

## 2 Higher-order linearisability

We examine higher-order libraries interacting with their context by means of abstract and public methods. In particular, we shall rely on types given by the grammar on the left below. We let \( \text{Meths} \) stand for the set of method names and assume \( \text{Meths} = \bigcup_{\theta \in \Theta} \text{Meths}_{\theta, \theta'} \), where each set \( \text{Meths}_{\theta, \theta'} \) contains names for methods of type \( \theta \to \theta' \). Methods are ranged over by \( m \) (and variants). We let \( v \) range over computational values, which include a unit value, integers, methods and pairs of values.

\[
\theta ::= \text{unit} \mid \text{int} \mid \theta \times \theta \mid \theta \\
v ::= (\_ \mid i \mid \text{int} \mid (v, v))
\]

The framework of a higher-order library and its environment is depicted in Figure 1. Given \( \Theta, \Theta' \subseteq \text{Meths} \), a library \( L \) is said to have type \( \Theta \to \Theta' \) if it defines public methods with names (and types) as in \( \Theta' \), using abstract methods \( \Theta \). The environment of \( L \) consists of a client \( K \) (which invokes public methods of \( \Theta' \)), and a parameter library \( L' \) (which provides code for the abstract methods \( \Theta \)). In general, \( K \) and \( L' \) may interact via a disjoint set of methods \( \Theta'' \subseteq \text{Meths} \), to which \( L \) has no access.
In the rest of this paper, we shall implicitly assume that we work with a library $L$ operating in an environment presented in Figure 1. The client $K$ will consist of a fixed number $N$ of concurrent threads. Next we introduce a notion of history tailored to the setting and define how histories can be linearised. In Section 3 we present the syntax for libraries and clients, and in Section 4 we define their semantics in terms of histories and co-histories respectively.

2.1 Higher-order histories

The operational semantics of libraries will be given in terms of histories, which are sequences of method calls and returns, each decorated with a thread identifier $t$ and a polarity index $XY$, where $X \in \{O,P\}$ and $Y \in \{L,K\}$, as shown below.

$$(t,\text{call } m(v))_{XY} \quad (t,\text{ret } m(v))_{XY}$$

We shall refer such decorated calls and returns as moves. Here, $m$ is a method name and $v$ is a value of a matching type. The index $XY$ specifies which of the three entities $(L,L',K)$ produces the move, and towards whom it is addressed.

- If $XY = PL$ then the move is issued by $L$, and is addressed to $L'$.
- If $XY = PK$ then the move is issued by $L$, and is addressed to $K$.
- If $XY = PL'$ then the move is issued by $L'$, and is addressed to $L$.
- If $XY = OK$ then the move is issued by $K$, and is addressed to $L$.

The choice of indices is motivated by the fact that the moves can be seen as defining a 2-player game between the library $(L)$, and its environment $(L',K)$ that represents the Opponent $(O)$. Moves played between $L$ and $L'$ are moreover decorated with $L$, whereas those between $L$ and $K$ have $K$ instead. Note that any potential interaction between $L'$ and $K$ is invisible to $L$ and is therefore not accounted for in the game (but we will later see how it can affect it). We use $O$ to refer to either $OK$ or $OL$, and $P$ to refer to either $PK$ or $PL$. Finally, we let the dual polarity of $XY$ to be $X'Y'$, where $X \not= X'$. For example, the dual of $PL$ is $OL$.

Next we proceed to define histories. Their definition will rely on a more primitive concept of prehistories, which are sequences of method calls and returns that respect a stack discipline.

► Definition 2. Prehistories are sequences generated by one of the grammars:

$$\begin{align*}
\text{PreH}_O & ::= \epsilon \mid \text{call } m(v)_{OY} \text{PreH}_P \text{ ret } m(v')_{PY} \text{PreH}_O \\
\text{PreH}_P & ::= \epsilon \mid \text{call } m(v)_{PY} \text{PreH}_O \text{ ret } m(v')_{OY} \text{PreH}_P
\end{align*}$$

where, in each line, the two occurrences of $Y \in \{K,L\}$ and $m \in \text{Meths}$ must each match. Moreover, if $m \in \text{Meths}_{\theta,\theta'}$, the types of $v, v'$ must match $\theta, \theta'$ respectively. We let $\text{PreH} = \text{PreH}_O \cup \text{PreH}_P$.

Thus, prehistories from $\text{PreH}_O$ start with an O-move, while those in $\text{PreH}_P$ start with a P-move. In each case, the polarities inside a prehistory alternate between $O$ and $P$, and the polarities of calls and matching returns are always dual (returns dual to calls). For example, a call made by $L$ to $L'$ (tagged $PL'$) must be matched by a return from $L'$ to $L$ (tagged $OL$).

Histories will be interleavings of prehistories tagged with thread identifiers (natural numbers), subject to a set of well-formedness constrains. In particular, a history $h$ for library $L : \Theta \to \Theta'$ will have to begin with an $O$-move and satisfy the following conditions, to be formalised in Definition 3.
1. The name of any method called in \( h \) must come from \( \Theta \) or \( \Theta' \), or be introduced earlier in \( h \) as a higher-order argument or result (no methods out of thin air). In addition:
   - if the method is from \( \Theta' \), the call must be tagged with \( \text{OK} \) (i.e. issued by \( K \));
   - if the method is from \( \Theta \), the call must be tagged with \( \text{PL} \) (i.e. issued by \( L \) towards \( L' \));
   - for a call of method \( m \notin \Theta \cup \Theta' \) to be valid, \( m \) must be introduced in an earlier move of dual polarity (calls dual to introductions).

2. Any method name appearing inside a call or return argument in \( h \) must be fresh, i.e. not used earlier. This reflects the assumption that methods can be called and returned, but not compared for identity (introductions always fresh).

Given \( h \in \text{PreH} \) and \( t \in \mathbb{N} \), we write \( t \times h \) for \( h \) in which each call or return is decorated with \( t \). We refer to such moves with \((t, \text{call} \ m(v))_{XY} \) or \((t, \text{ret} \ m(v))_{XY} \) respectively. If we only want to stress the \( X \) or \( Y \) membership, we shall drop \( Y \) or \( X \) respectively. Moreover, when no confusion arises, we may sometimes drop a move’s polarity altogether. We say that a move \( x \) introduces a name \( m \in \text{Meths} \) when \( x \in \{ \text{call} \ m'(v), \text{ret} \ m'(v) \} \) for some \( m', v \) such that \( v \) contains \( m \).

\( \triangleright \) **Definition 3.** Given \( \Theta, \Theta' \), the set of histories over \( \Theta \to \Theta' \), written \( \mathcal{H}_{\Theta, \Theta'} \), is defined by

\[
\mathcal{H}_{\Theta, \Theta'} = \bigcup_{N>0} \bigcup_{h_1, \ldots, h_N \in \text{PreH}_L} (1 \times h_1) \cdots (N \times h_N)
\]

where \((1 \times h_1) \cdots (N \times h_N)\) is the set of all interleavings of \((1 \times h_1), \ldots, (N \times h_N)\) satisfying:

1. For any \( s_1(t, \text{call} \ m(v))_{XY} s_2 \in \mathcal{H}_{\Theta, \Theta'} \), either \( m \in \Theta' \) and \( XY = \text{OK} \), or \( m \in \Theta \) and \( XY = \text{PL} \), or there is a move \((t', x)_{XY}^Y \) in \( s_1 \) such that \( X \neq X' \) and \( x \) introduces \( m \).

2. For any \( s_1(t, x)_{XY} s_2 \in \mathcal{H}_{\Theta, \Theta'} \) and any \( m \), if \( m \) is introduced by \( x \) then \( m \) must not occur in \( s_1 \).

A history \( h \in \mathcal{H}_{\Theta, \Theta'} \) is called **sequential** if it is of the form

\[
h = (t_1, x_1)_{OY_1} (t_1, x'_1)_{PY_1} \cdots (t_k, x_k)_{OY_k} (t_k, x'_k)_{PY_k}
\]

for some \( t_i, x_i, x'_i, Y_i, Y'_i \). We write \( \mathcal{H}_{\Theta, \Theta'}^{\text{seq}} \) for the set of all sequential histories from \( \mathcal{H}_{\Theta, \Theta'} \).

We shall range over \( \mathcal{H}_{\Theta, \Theta'} \) using \( h, s \) (and variants). The subscripts \( \Theta, \Theta' \) will often be omitted. Given a history \( h \), we shall write \( \overline{h} \) for the sequence of moves obtained from \( h \) by dualising all move polarities inside it. The set of co-histories over \( \Theta \to \Theta' \) will be \( \mathcal{H}_{\Theta, \Theta'}^{\text{co}} = \{ \overline{h} | h \in \mathcal{H}_{\Theta, \Theta'} \} \).

While in this section histories will be extracted from example libraries informally, in Section 4 we give the formal semantics \([L]\) of libraries. For each \( L : \Theta \to \Theta' \), we shall have \([L] \subseteq \mathcal{H}_{\Theta, \Theta'} \).

\( \triangleright \) **Remark 4.** The notion of history introduced above extends the classic notion from [12] to higher-order types. It also extends the notion presented in [3]. The intuition behind the definition is that a history is a sequence of (well-bracketed) method calls and returns, called moves, each tagged with a thread identifier and a polarity, where polarities track the originators and recipients of moves. Moves may be calls or returns related to methods given in the library interface \((\Theta \to \Theta')\), or dynamically created methods that appear earlier inside the histories—recall that, in a higher-order setting, methods can be passed around as arguments to calls or be returned as results by other methods. On the other hand, a sequential history is one in which the operations performed by the library can be perceived as atomic, that is, each move produced by \( O \) is to be immediately followed by the library’s response, which is a \( P \) move in the same thread.
We present three notions of linearisability. First introduce a general notion that generalises classic linearisability \cite{12} and parameterised linearisability \cite{3}. We then develop two more specialised variants: a notion of encapsulated linearisability, following \cite{3}, that captures scenarios where the parameter library and the client cannot directly interact; and a relational notion whereby context behaviour (client and parameter library) is known to be relationally invariant.

We begin by introducing a class of reorderings on histories. Suppose $X, X’ \in \{O, P\}$ and $X \neq X’$. We let $\prec_{XX'} \subseteq \mathcal{H}_{\Theta, \Theta} \times \mathcal{H}_{\Theta, \Theta'}$ be the smallest binary relation over $\mathcal{H}_{\Theta, \Theta'}$ satisfying, for any $t \neq t'$:

\[ s_1(t', x') \prec_{XX'} (t, x) Z s_2 \quad \prec_{XX'} s_1(t, x) Z (t', x') Z s_2 \]
whenever $Z = X$ or $Z' = X'$. Intuitively, two histories $h_1, h_2$ are related by $\sim_{XX'}$ if the latter can be obtained from the former by swapping two adjacent moves from different threads in such a way that, after the swap, an $X$-move will occur earlier or an $X'$-move will occur later. Note that, because of $X \neq X'$, the relation always applies to adjacent moves of the same polarity. On the other hand, we cannot have $s_1(t,x)X(t',x')Xs_2 \sim_{XX'} s_1(t',x')X(t,x)s_2$.

**Definition 6 (General Linearisability).** Given $h_1, h_2 \in \mathcal{H}_{\Theta, \Theta'}$, we say that $h_1$ is linearised by $h_2$, written $h_1 \triangleleft_{\Theta, \Theta'} h_2$, if $h_1 \sim_{\Theta, \Theta'} h_2$. Given libraries $L, L' : \Theta \to \Theta'$ and a set of sequential histories $A \subseteq \mathcal{H}^\seq_{\Theta, \Theta'}$, we write $L \triangleleft_{\Theta, \Theta'} A$, and say that $L$ can be linearised to $A$, if for any $h \in [L]$ there exists $h' \in A$ such that $h \sim h'$. Moreover, we write $L \triangleleft_{\Theta, \Theta'} L'$ if $L \subseteq [L'] \cap \mathcal{H}^\seq_{\Theta, \Theta'}$ (i.e. for all $h \in [L]$ there is sequential $h' \in [L']$ such that $h \sim h'$).

**Remark 7.** The classic notion of linearisability from [12] states that $h$ linearises to $h'$ just if the return/call order of $h$ is preserved in $h'$ (and $h'$ is sequential), i.e. if a return move precedes a call move in $h$ then so is the case in $h'$. Observing that, in [12], return and call moves coincide with $P$- and $O$-moves respectively, we can see that our higher-order notion of linearisability is a generalisation of the classic notion.

We next show that a more permissive notion of linearisability applies if the parameter library $L'$ of Figure 1 is encapsulated, that is, the client $K$ can have no direct access to it (i.e. $\Theta' = \emptyset$). To capture the more restrictive nature of interaction, we introduce a more constrained notion of a history. Specifically, in addition to sequentiality in every thread, we shall insist that a move made in the library in the $L$ or $K$ component must be followed by an $O$ move from the same component.

**Definition 8.** We call a history $h \in \mathcal{H}_{\Theta, \Theta'}$ encapsulated if, for each thread $t$, we have that if $h = s_1(t,x)_{P,Y} s_2(t,x')_{O,Y} s_3$ and moves from $t$ are absent from $s_2$ then $Y = Y'$. Moreover, we set $\mathcal{H}^\enc_{\Theta, \Theta'} = \{ h \in \mathcal{H}_{\Theta, \Theta'} \mid h$ encapsulated $\}$ and $[L]^\enc = [L] \cap \mathcal{H}^\enc_{\Theta, \Theta'}$ (if $L : \Theta \to \Theta'$).

We define the corresponding linearisability notion as follows. First, let $\circ \subseteq \mathcal{H}_{\Theta, \Theta'} \times \mathcal{H}_{\Theta, \Theta'}$ be the smallest binary relation on $\mathcal{H}_{\Theta, \Theta'}$ such that, for any $Y, Y' \in \{ K, L \}$ with $Y \neq Y'$ and $t \neq t'$:

$$s_1(t,m)Y(t',m')Y' \circ s_1(t',m')Y(t,m)Y_2$$

**Definition 9 (Encapsulated linearisability).** Given $h_1, h_2 \in \mathcal{H}^\enc_{\Theta, \Theta'}$, we say that $h_1$ is enc-linearised by $h_2$, and write $h_1 \triangleleft_{\Theta, \Theta'} h_2$, if $h_1(\angle_{PO} \cup \circ)^* h_2$ and $h_2$ is sequential. A library $L : \Theta \to \Theta'$ can be enc-linearised to $A$, written $L \triangleleft_{\Theta, \Theta'} A$, if $A \subseteq \mathcal{H}^\enc_{\Theta, \Theta'} \cap \mathcal{H}^\seq_{\Theta, \Theta'}$, and for any $h \in [L]^\enc$ there exists $h' \in A$ such that $h \triangleleft_{\Theta, \Theta'} h'$. We write $L \triangleleft_{\Theta, \Theta'} L'$ if $L \subseteq [L']^\enc \cap \mathcal{H}^\seq_{\Theta, \Theta'}$.

**Remark 10.** Suppose $\Theta = \{ m : \text{int} \to \text{int} \}$ and $\Theta' = \{ m' : \text{int} \to \text{int} \}$. Histories from $\mathcal{H}_{\Theta, \Theta'}$ may contain the following actions only: call $m(i)_O, \text{ret } m(i)_O, \text{call } m(i)_P, \text{call } m'(i)_P$. Then $(\angle_{PO} \cup \circ)^*$ preserves the order between call $m(i)_P$ and call $m(i)_O$ as well as that between call $m'(i)_P$ and call $m'(i)_O$, i.e. it coincides with Definition 3 of [3].

**Example 11 (Parameterised multiset).** We revisit the multiset library of Example 1 and extend it with a public method $\text{reset}$, which performs multiplicity resets to default values using an abstract method $\text{default}$ as the default-value function (again, we use absolute values to avoid negative multiplicities). The extended library is shown in Figure 3 and written $L_{\text{mult}} : \{ \text{default} \} \to \Theta'$, with $\Theta' = \{ \text{count}, \text{update}, \text{reset} \}$. In contrast to the $\text{update}$ method of $L_{\text{mult}}$, $\text{reset}$ is not optimistic: it retrieves the lock upon its call, and only releases it before return. In particular, the method calls $\text{default}$ while it retains the lock.
Figure 3 Left: Parameterised multiset library $L_{\text{mset2}}$ (lines 5-19 as in Fig. 2) with public methods $\text{count}$, $\text{reset}$: $\text{int} \rightarrow \text{int}$, $\text{update}$: $\text{int} \times (\text{int} \rightarrow \text{int}) \rightarrow \text{int}$; abstract method $\text{default}$: $\text{int} \rightarrow \text{int}$. Right: Flat combination library $L_b$.

Observe that, were $\text{default}$ able to externally call $\text{update}$, we would reach a deadlock: $\text{default}$ would be keeping the lock while waiting for the return of a method that requires the lock. On the other hand, if the library is encapsulated then the latter scenario is not possible. In such a case, $L_{\text{mset2}}$ linearises to the specification $A_{\text{mset2}}$, defined next. Let $A_{\text{mset2}} = \{ \pi_1(s) \mid s \in A_{\text{mset2}} \}$ where:

$$A_{\text{mset2}} = \{ s \in H_{\Theta, \Theta'} | \pi_1(s) \in H_{\Theta, \Theta'}^{\text{seq}} \land \forall t, s \vdash t \in S \land \forall s'(\_, I) P(\_, J) \in \text{pre} s, I = J \}$$

and the set $S$ is now given by the grammar of Example 5 extended with the rule:

$$S \rightarrow (t, \text{call reset}(i), \text{OK}) (t, \text{call default}(i), \text{I}_{\text{PC}}) (t, \text{ret default}(j), \text{I}_{\text{OC}}) (t, \text{ret reset}(j), \text{I'}_{\text{PC}} S$$

with $I' = I[i \mapsto [j]]$. Our framework makes it possible to confirm that $L_{\text{mset2}}$ enc-linearises to $A_{\text{mset2}}$.

We finally extend general linearisability to cater for situations where the client and the parameter library adhere to closure constraints expressed by relations $R$ on histories. Let $\Theta, \Theta'$ be sets of abstract and public methods respectively. The closure relations we consider are closed under permutations of methods outside $\Theta \cup \Theta'$: if $h_1 R h'$ and $\pi$ is a (type-preserving) permutation on $\text{Meths} \setminus (\Theta \cup \Theta')$ then $\pi(h) R \pi(h')$. The requirement represents the fact that, apart from the method names from a library interface, the other method names are arbitrary and can be freely permuted without any observable effect. Thus, $R$ should not be distinguishing between such names.

Definition 12 (Relational linearisability). Let $R \subseteq H_{\Theta, \Theta'} \times H_{\Theta, \Theta'}$ be closed under permutations of names in $\text{Meths} \setminus (\Theta \cup \Theta')$. Given $h_1, h_2 \in H_{\Theta, \Theta'}$, we say that $h_1$ is $R$-linearised by $h_2$, and write $h_1 \in R h_2$, if $h_1(\pi \circ R)^* h_2$ and $h_2$ is sequential. A library $L : \Theta \rightarrow \Theta'$ can be $R$-linearised to $A$, written $L \in R A$, if $A \subseteq H_{\Theta, \Theta'}^{\text{seq}}$ and for any $h \in [L]$ there exists $h' \in A$ such that $h \in R h'$. We write $L \in R L'$ if $L \subseteq R [L'] \cap H_{\Theta, \Theta'}^{\text{seq}}$.

Example 13. We consider a higher-order variant of an example from [3] that motivates relational linearisability. Flat combining [11] is a synchronisation paradigm that advocates
the use of a single thread holding a global lock to process requests of all other threads. To facilitate this, threads share an array to which they write the details of their requests and wait either until they acquire a lock or their request has been processed by another thread. Once a thread acquires a lock, it executes all requests stored in the array and the outcomes are written to the array for access by the requesting threads.

Let $\Theta' = \{ \text{run} \in \text{Meth}_\text{ref}(\theta \to \theta') \}$, where $\text{Meth}_\text{ref}$ is the set of method references (Figure 3, right). The library $L\text{fc} : \emptyset \to \Theta'$ is built following the flat combining approach and, on acquisition of the global lock, the winning thread acts as a combiner of all registered requests. Note that the requests will be attended to one after another (thus guaranteeing mutual exclusion) and only one lock acquisition will suffice to process one array of requests. Using our framework, one can show that $L\text{fc}$ can be $R$-linearised to the specification given by the library $L\text{spec}$ defined by

$$
\text{run} = \lambda (f, x). (\text{lock.acquire}(); \text{let result} = f(x) \text{ in lock.release}(); \text{result})
$$

where each function call in $L\text{spec}$ is protected by a lock. Observe that we cannot hope for $L\text{fc} \subseteq L\text{spec}$, because clients may call library methods with functional arguments that recognise thread identity. Consequently, we can relate the two libraries only if context behaviour is guaranteed to be independent of thread identifiers. This can be expressed through $\in R$, where $R \subseteq H_{\Theta, \Theta'} \times H_{\Theta, \Theta'}$ is a relation capturing thread-blind client behaviour.

### 3 Library syntax

We now look at the concrete syntax of libraries and clients. Libraries comprise collections of typed methods whose argument and result types adhere to the grammar: $\theta ::= \text{unit} | \text{int} | \theta \to \theta$. We shall use three disjoint enumerable sets of names, referred to as $\text{Vars}$, $\text{Meths}$ and $\text{Refs}$, to name respectively variables, methods and references. $x, f$ (and their decorated variants) will be used to range over $\text{Vars}$; $m$ will range over $\text{Meths}$; and $r$ over $\text{Refs}$. Methods and references are implicitly typed, i.e. $\text{Meths} = \bigcup_{\theta, \theta'} \text{Meths}_{\theta, \theta'}$ and $\text{Refs} = \bigcup_{\theta, \theta'} \text{Refs}_{\theta, \theta'}$, where $\text{Meths}_{\theta, \theta'}$ contains names for methods of type $\theta \to \theta'$, $\text{Refs}_{\theta, \theta'}$ contains names for integer references and $\text{Refs}_{\theta, \theta'}$ contains names for references to methods of type $\theta \to \theta'$. We write $\cup$ for disjoint set union.

The syntax for libraries and clients is given in Figure 4. Each library $L$ begins with a series of method declarations (public or abstract) followed by a block $B$ containing method implementations ($m = \lambda x.M$) and reference initialisations ($r := i$ or $r := \lambda x.M$). The typing rules ensure that each public method is implemented within the block, in contrast to abstract methods. Clients are parallel compositions of closed terms.

Terms $M$ specify the shape of allowable method bodies. $()$ is the skip command, $i$ ranges over integers, $t_{\text{id}}$ is the current thread identifier and $\emptyset$ represents standard arithmetic operations. Thanks to higher-order references, we can simulate divergence by $(!r)()$, where $r \in \text{Refs}_{\text{unit}, \text{unit}}$ is initialised with $\lambda x.\text{unit}.(!r)()$. Similarly, while $M \cdot N$ can be simulated by $(!r)()$ after $r := \lambda x.\text{unit}.let y = M \text{ in (if } y \text{ then } N; (!r)()) \text{ else } ()$. We also use the standard derived syntax for sequential composition, i.e. $M; N$ stands for let $x = M$ in $N$, where $x$ does not occur in $N$. For each term $M$, we write $\text{Meths}(M)$ for the set of method names occurring in $M$. We use the same notation for method names in blocks and libraries.

**Remark 14.** In Section 2 we used lock-related operations in our example libraries ($\text{acquire}$, $\text{tryacquire}$, $\text{release}$), on the understanding that they can be coded using shared memory. Similarly, the array of Example 13 in the sequel can be constructed using references.

Also, the initialisation statements must comply with types.

The semantics of our system is given in several stages. First, we define an operational semantics for sequential and concurrent terms that may draw methods from a repository. We then adapt it to capture interactions of concurrent clients with closed libraries (no abstract methods). This notion is then used to define contextual approximation for arbitrary libraries. Finally, we introduce a trace semantics of arbitrary libraries, which generates the histories on which our notions of linearisability are based.

\begin{itemize}
\item **Figure 4** Library syntax, and typing rules for terms (\(\vdash\)), blocks (\(\vdash_B\)), libraries (\(\vdash_L\)), clients (\(\vdash_K\)).
\end{itemize}

For simplicity, we do not include private methods, yet the same effect could be achieved by storing them in higher-order references. As we explain in the next section, references present in library definitions are de facto private to the library. Note also that, according to our definition, sets of abstract and public methods are disjoint. However, given \(m, m' \in \text{Refs}_{\theta', \theta}\), one can define a “public abstract” method with: public \(m;\) abstract \(m'\); \(m = \lambda x. m' x\).

Terms are typed in environments \(\Gamma = \{x_1 : \theta_1, \ldots, x_n : \theta_n\}\). Method blocks are typed through judgements \(\vdash_B B : \Theta\), where \(\Theta \subseteq \text{Meths}\). The judgments collect the names of methods defined in a block as well as making sure that the definitions respect types and are not duplicated. Also, the initialisation statements must comply with types.

Finally, we type libraries using statements of the form \(\Theta \vdash_L L : \Theta' \rightarrow \Theta''\), where \(\Theta, \Theta', \Theta'' \subseteq \text{Meths}\) and \(\Theta' \cap \Theta'' = \emptyset\). The judgment \(\emptyset \vdash_L L : \Theta' \rightarrow \Theta''\) guarantees that any method occurring in \(L\) is present either in \(\Theta'\) or \(\Theta''\), that all methods in \(\Theta'\) are declared as abstract and unimplemented, while all methods in \(\Theta''\) are declared as public and defined. Thus, \(\emptyset \vdash_L L : \Theta \rightarrow \Theta'\) is a library in which \(\Theta, \Theta'\) are the abstract and public methods respectively. In this case, we also write \(L : \Theta \rightarrow \Theta'\).

\section{Semantics and soundness}

The semantics of our system is given in several stages. First, we define an operational semantics for sequential and concurrent terms that may draw methods from a repository. We then adapt it to capture interactions of concurrent clients with closed libraries (no abstract methods). This notion is then used to define contextual approximation for arbitrary libraries. Finally, we introduce a trace semantics of arbitrary libraries, which generates the histories on which our notions of linearisability are based.
Higher-Order Linearisability

\[(L) \rightarrow_{\text{lib}} (L, \emptyset, S_{\text{init}})\]
\[(r := i; B, \mathcal{R}, S) \rightarrow_{\text{lib}} (B, \mathcal{R}, S[r \mapsto i])\]
\[(\text{abstract } m; L, \mathcal{R}, S) \rightarrow_{\text{lib}} (L, \mathcal{R}, S)\]
\[(m = \lambda x.M; B, \mathcal{R}, S) \rightarrow_{\text{lib}} (B, \mathcal{R}_{++}, S)\]
\[(\text{public } m; L, \mathcal{R}, S) \rightarrow_{\text{lib}} (L, \mathcal{R}, S)\]
\[(r := \lambda x.M; B, \mathcal{R}, S) \rightarrow_{\text{lib}} (B; \mathcal{R}_{++}, S[r \mapsto m])\]

\[E = \bullet | E \oplus M | i \oplus E | \text{if } E \text{ then } M \text{ else } M | \pi_j E | (E, M) | (v, E) | mE | \text{let } x = E \text{ in } M | r := E\]

\[M, \mathcal{R}, S \rightarrow_i (M', \mathcal{R}', S')\]

\[\text{Evaluating rules for } E \text{ (}\rightarrow_{\text{lib}}\text{), terms (}\rightarrow_i\text{) and clients (}\rightarrow\text{). In the rules above we use the conditions/notation: }\mathcal{R}_{++} = \mathcal{R} \oplus (m \mapsto \lambda x.M), i_{++} = i_1 \oplus i_2, R_x(m) = \lambda x.M, \text{ and } j_x = 0 \text{ iff } i_x = 0.\]

4.1 Library-client evaluation

Libraries, terms and clients are evaluated in environments comprising:

- A method environment \(\mathcal{R}\), called \(\text{own-method repository}\), which is a finite partial map on 
  \(\text{Meths}\) assigning to each \(m\) in its domain, with \(m \in \text{Meths}_{\theta, \omega}\), a term of the form \(\lambda y.M\) (we omit type-superscripts from bound variables for economy).

- A finite partial map \(S : \text{Refs} \rightarrow (\mathbb{Z} \cup \text{Meths})\), called \(\text{store}\), which assigns to each \(r\) in its domain an integer (if \(r \in \text{Refs}_{\text{init}}\)) or name from \(\text{Meths}_{\theta, \omega}\) (if \(r \in \text{Refs}_{\text{lib}, \theta, \omega}\)).

The evaluation rules are presented in Figure 5, where we also define \(\text{evaluation contexts} \ E\).

\[\text{Remark 15. We shall assume that reference names used in libraries are library-private, i.e. sets of reference names used in different libraries are assumed to be disjoint. Similarly, when libraries are being used by client code, this is done on the understanding that the references available to that code do not overlap with those used by libraries. Still, for simplicity, we shall rely on a single set } \text{Refs} \text{ of references in our operational rules.}\]

First we evaluate the library to create an initial repository and store. This is achieved by the first set of rules in Figure 5, where we assume that \(S_{\text{init}}\) is empty. Thus, library evaluation produces a tuple \((\epsilon, \mathcal{R}_0, S_0)\) including a method repository and a store, which can be used as the initial repository and store for evaluating \(M_0 \mathcal{R}_1 \cdots M_N\) using the \((K_N)\) rule.

We shall call the latter evaluation semantics for clients (denoted by \(\rightarrow\)) the \(\text{multi-threaded operational semantics}\). The latter relies on closed-term reduction \((\rightarrow_i)\), whose rules are given in the middle group, where \(t\) is the current thread index. Note that the rules for \(E[\lambda x.M]\) in the middle group, along with those for \(m = \lambda x.M\) and \(r := \lambda x.M\) in the first group, involve the creation of a fresh method name \(m\), which is used to put the function in the repository \(\mathcal{R}\). Name creation is non-deterministic: any fresh \(m\) of the appropriate type can be chosen.

We define termination for clients linked with libraries that have no abstract methods. Recall our convention (Remark 15) that \(L\) and \(M_1, \cdots, M_N\) must access disjoint parts of the store. Terms \(M_1, \cdots, M_N\) can share reference names, though.
We shall build a notion of contextual approximation of libraries on top of termination: one library will be said to approximate another if, whenever the former terminates when composed with any parameter library and client, so does the latter.

We will be considering the following notions for composing libraries. Let us denote a library $L$ as $L = D; B$, where $D$ contains all the (public/abstract) method declarations of $L$, and $B$ is its method block. We write $\text{Refs}(L)$ for the set of references in $L$. Let $L_1 : \Theta_1 \rightarrow \Theta_2$ be of the form $D_1; B_1$. Given $L_2 : \Theta'_1 \rightarrow \Theta'_2 (= D_2; B_2)$ such that $\Theta_2 \cap \Theta'_2 = \text{Refs}(L_1) \cap \text{Refs}(L_2) = \emptyset$, $\Theta = \{m_1, ..., m_n\} \subseteq \Theta_2$ and $L' : \Theta \rightarrow \Theta_1, \Theta'$, we define the union of $L_1$ and $L_2$, the $\Theta$-hiding of $L_1$, and the sequencing of $L'$ with $L_1$ respectively as:

\[
L_1 \cup L_2 : (\Theta_1 \cup \Theta'_1) \setminus (\Theta_2 \cup \Theta'_2) \rightarrow \Theta_2 \cup \Theta'_2 = (D_1; B_1) \cup (D_2; B_2) = D'_1; D'_2; B_1; B_2
\]

\[
L_1 \setminus \Theta : \Theta_1 \rightarrow (\Theta \setminus \Theta)
\]

\[
L'; L_1 : \emptyset \rightarrow \Theta_2, \Theta'
\]

where $D'_1$ is $D_1$ with any abstract $m$ declaration removed for $m \in \Theta'_2$, dually for $D'_2$; and where $D''_2$ is $D_2$ without public $m$ declarations for $m \in \Theta$ and each $r_i$ is a fresh reference matching the type of $m_i$, and $B'_1$ is obtained from $B_1$ by replacing each $m_i = \lambda x.M$ by $r_i := \lambda x.M$. Thus, the union of $L_1$ and $L_2$ corresponds to merging their code and removing any abstract declarations for methods that become defined. The hiding of a public method simply renders it private via the use of references.

Definition 17. Given $L_1, L_2 : \Theta \rightarrow \Theta'$, we say that $L_1$ contextually approximates $L_2$, written $L_1 \sqsubseteq L_2$, if for all $L' : \emptyset \rightarrow \Theta, \Theta''$ and $\Theta', \Theta'' \rightarrow \emptyset, M_1 \ldots \| M_N : \text{unit}$, if link $L'$; $L_1$ in $(M_1 \ldots \| M_N) \downarrow$ then link $L' ; L_2$ in $(M_1 \ldots \| M_N) \downarrow$. In this case, we also say that $L_2$ contextually refines $L_1$.

Note that, according to this definition, the parameter library $L'$ may communicate directly with the client terms through a common interface $\Theta''$. We shall refer to this case as the general case. Later on, we shall also consider more restrictive testing scenarios in which this possibility of explicit communication is removed. Moreover, from the disjointness conditions in the definitions of sequencing and linking we have that $L_i$, $L'$ and $M_1 \ldots \| M_N$ access pairwise disjoint parts of the store.

4.2 Trace semantics

Building on the earlier semantics, we next introduce a trace semantics of libraries in the spirit of game semantics [2]. As mentioned in Section 2, the behaviour of a library will be represented as an exchange of moves between two players called $P$ and $O$, representing the library and its corresponding context respectively. The context consists of the client of the library as well as the parameter library, with an index on each move $(K/L)$ specifying which of them is involved in the move.

In contrast to the previous section, we handle scenarios in which called methods need not be present in the repository $R$. Calls to such undefined methods are represented by labelled transitions calls to the context made on behalf of the library ($P$). The calls can later be responded to with labelled transitions corresponding to returns, made by the context ($O$). On the other hand, $O$ is able to invoke methods in $R$, which will also be represented through suitable labels. Because we work in a higher-order setting, calls and returns made by both players may involve methods as arguments or results. Such methods also become available.
Higher-Order Linearisability

\[(\text{Int}) \quad (\mathcal{E}, M, R, P, A, S) \rightarrow_t (\mathcal{E}, M', R', P, A, S'), \quad \text{given that } (M, R, S) \rightarrow_t (M', R', S') \text{ and } \text{dom}(R' \setminus R) \text{ consists of names that do not occur in } \mathcal{E}, A.\]

\[(\text{PQy}) \quad (\mathcal{E}, E[mv], R, P, A, S) \xrightarrow{\text{call } m(v')} (m : E : \mathcal{E}, R', P', A, S), \quad \text{given } m \in A_Y \text{ and } (\text{PC}).\]

\[(\text{OQy}) \quad (\mathcal{E}, E : \_, R, P, A, S) \xrightarrow{\text{call } m(v)} (m : \mathcal{E}, M\{v/x\}, R, P, A', S), \quad \text{given } m \in P_Y, \quad R(m) = \lambda x.M \text{ and } (\text{OC}).\]

\[(\text{PAy}) \quad (m : \mathcal{E}, v, R, P, A, S) \xrightarrow{\text{ret } m(v')} (\mathcal{E}, E[v], R', P', A, S), \quad \text{given } m \in P_Y \text{ and } (\text{PC}).\]

\[(\text{OAy}) \quad (m : \mathcal{E}, E : \_, R, P, A, S) \xrightarrow{\text{ret } m(v)} (\mathcal{E}, E[v], R, P, A', S), \quad \text{given } m \in A_Y \text{ and } (\text{OC}).\]

\[(\text{PC}) \quad \text{If } v \text{ contains the names } m_1, \ldots, m_k \text{ then } v' = v\{m_i'[m_i] \mid 1 \leq i \leq k\} \text{ with each } m'_i \text{ being a fresh name. Moreover, } R' = R \cup \{m'_i \mapsto \lambda x.m_i x \mid 1 \leq i \leq k\} \text{ and } P' = P \cup \{m_1, \ldots, m_k\}.\]

\[(\text{OC}) \quad \text{If } v \text{ contains names } m_1, \ldots, m_k \text{ then } m_i \in \phi(P, A), \text{ for each } i, \text{ and } A' = A \cup \{m_1, \ldots, m_k\}.\]

\section*{Figure 6} Trace semantics rules. The rule \((\text{Int})\) is for embedding internal rules. In the rule \((\text{PQy})\), the library \((P)\) calls one of its abstract methods (either the original ones or those acquired via interaction), while in \((\text{PAy})\) it returns from such a call. The rules \((\text{OQy})\) and \((\text{OAy})\) are dual and represent actions of the context. In all of the rules, whenever we write \(m(v)\) or \(m(v')\), we assume that the type of \(v\) matches the argument type of \(m\).

for future calls: function arguments/results supplied by \(P\) are added to the repository and can later be invoked by \(O\), while function arguments/results provided by \(O\) can be queried in the same way as abstract methods.

The trace semantics utilises configurations that carry more components than the previous semantics. We define two kinds of configurations:

\begin{itemize}
    \item \textit{O-configurations} \((\mathcal{E}, \_, \_, R, P, A, S)\)
    \item \textit{P-configurations} \((\mathcal{E}, M, R, P, A, S)\)
\end{itemize}

where the component \(\mathcal{E}\) is an \textit{evaluation stack}, that is, a stack of the form \([X_1, X_2, \ldots, X_n]\) with each \(X_i\) being either an evaluation context or a method name. On the other hand, \(P = (P_L, P_K)\) with \(P_L, P_K \subseteq \text{dom}(R)\) being sets of \textit{public} method names, and \(A = (A_L, A_K)\) is a pair of sets of \textit{abstract} method names. \(P\) will be used to record all the method names produced by \(P\) and passed to \(O\); those passed to \(OK\) are stored in \(P_K\), while those leaked to \(OL\) are kept in \(P_L\). Inside \(A\), the story is the opposite one: \(A_K\) stores the method names produced by \(OK\) (resp. \(OL\)) and passed to \(P\). Consequently, the sets of names stored in \(P_L, P_K, A_L, A_K\) will always be disjoint.

Given a pair \(P\) as above and a set \(Z \subseteq \text{Meths}\), we write \(P \cup K \cup Z\) for the pair \((P_L, P_K \cup Z)\). We define \(\cup\) in a similar manner, and extend it to pairs \(A\) as well. Moreover, given \(P\) and \(A\), we let \(\phi(P, A)\) be the set of \textit{fresh} method names for \(P, A\): \(\phi(P, A) = \text{Meths} \setminus (P_L \cup P_K \cup A_L \cup A_K)\).

We give the rules generating the trace semantics in Figure 6. Note that the rules are parameterised by: \(P/O\) and \(Y\), which together determine the polarity of the next move; \(Q/A\), which stands for the move being a call (\textit{Question}) or a return (\textit{Answer}) respectively. The rules depict the intuition presented above. When in an \textit{O}-configuration, the context may issue a call to a public method \(m \in P_Y\) and pass control to the library (rule \((\text{OQy})\)). Note that, when this occurs, the name \(m\) is added to the evaluation stack \(\mathcal{E}\) and a \textit{P}-configuration is obtained. From there on, the library will compute internally using rule \((\text{Int})\), until: it either needs to evaluate an abstract method (i.e. some \(m' \in A_Y\)), and hence issues a call via rule \((\text{PQy})\); or it completes its computation and returns the call (rule \((\text{PAy})\)). Calls to abstract methods, on the other hand, are met either by further calls to public methods (via \((\text{OQy})\)), or by returns (via \((\text{OAy})\)).

Finally, we extend the trace semantics to a concurrent setting where a fixed number of \(N\)-many threads run in parallel. Each thread has separate evaluation stack and term
components, which we write as $C = (E, X)$ (where $X$ is a term or “$-$”). Thus, a configuration now is of the following form:

$$N\text{-configuration } (C_1 || \cdots || C_N, R, P, A, S)$$

where, for each $i$, $C_i = (E_i, X_i)$ and $(E_i, X_i, R, P, A, S)$ is a sequential configuration. We shall abuse notation a little and write $(C_i, R, P, A, S)$ for $(E_i, X_i, R, P, A, S)$. Also, below we write $\tilde{C}$ for $C_1 || \cdots || C_N$ and $\tilde{C}[i \mapsto C'] = C_1 || \cdots || C_{i-1} || C' || C_{i+1} || \cdots || C_N$ and, for economy, we use $RPAS$ to range over tuples $(R, P, A, S)$. The corresponding definition of contextual approximation is the general notion of contextual approximation (refinement) from Definition 17. In the

Definition 20.

Let $\Theta', \Theta'' \vdash_k M_1 \cdots M_N : \text{unit}$ and $L' : \emptyset \to \Theta, \Theta''$. We define:

$$[\text{link } L' \vdash \text{in } (M_1 || \cdots || M_N)] = \{ s | (\tilde{C}_0, R_0, (\emptyset, \Theta'), (\emptyset, \Theta''), S_0) \xrightarrow{\ast} \rho \wedge \rho \text{ final} \}$$

where $(L') \xrightarrow{\ast_{lib}} (\epsilon, R_0, S_0)$ and $\tilde{C}_0 = ([], M_1 \cdots [])$. We may write $[L]_N$ simply as $[L]$.

We are now able to revisit the linearisability claims anticipated in Examples 1, 11 and 13.

Definition 21.

For any $L : \Theta \to \Theta'$, $L' : \emptyset \to \Theta', \Theta''$ and $\Theta', \Theta'' \vdash_k M_1 \cdots M_N : \text{unit}$ we have $[L]_N \subseteq H_{\Theta, \Theta'}$ and $[\text{link } L' \vdash \text{in } (M_1 || \cdots || M_N)] \subseteq H_{\Theta, \Theta'}^\infty$.

4.3 Soundness

To conclude, we clarify in what sense all the notions of linearisability are sound. Recall the general notion of contextual approximation (refinement) from Definition 17. In the encapsulated case libraries are being tested by clients that do not communicate with the parameter library explicitly. The corresponding definition of contextual approximation is defined below.
Definition 22 (Encapsulated $\subseteq$). Given libraries $L_1, L_2 : \Theta \to \Theta'$, we write $L_1 \subseteq_{\text{enc}} L_2$ when, for all $L' : \emptyset \to \Theta$ and $\Theta' \supseteq M_1 \ldots M_N : \text{unit}$, if link $L' ; L_1 \in (M_1 \ldots \| M_N) \downarrow$ then link $L' ; L_2 \in (M_1 \ldots \| M_N) \downarrow$.

For relational linearisability, we need yet another notion that will link $R$ to contextual testing.

Definition 23. Let $\mathcal{R} \subseteq \mathcal{H}_{\Theta, \Theta'} \times \mathcal{H}_{\Theta, \Theta'}$ be a set closed under permutation of names in Meths $\setminus (\Theta \cup \Theta')$. We say that a context formed by $L'$ and $M_1, \ldots, M_N$ is $\mathcal{R}$-closed if, for any $h \in \llbracket \text{link } L' ; \, \, \, \text{in} \, \, \, (M_1 \ldots \| M_N) \rrbracket$, $\overline{R \mathcal{R} \overline{R}}$ implies $h' \in \llbracket \text{link } L' ; \, \, \, \text{in} \, \, \, (M_1 \ldots \| M_N) \rrbracket$. Given $L_1, L_2 : \Theta \to \Theta'$, we write $L_1 \subseteq_{\mathcal{R}} L_2$ if, for all $\mathcal{R}$-closed contexts formed from $L', M_1, \ldots, M_N$, whenever link $L' ; L_1 \in (M_1 \ldots \| M_N) \downarrow$ then we also have link $L' ; L_2 \in (M_1 \ldots \| M_N) \downarrow$.

Theorem 24 (Correctness).
1. $L_1 \subseteq L_2$ implies $L_1 \subseteq_{\text{enc}} L_2$.
2. $L_1 \subseteq_{\text{enc}} L_2$ implies $L_1 \subseteq_{\mathcal{R}} L_2$.
3. $L_1 \subseteq_{\mathcal{R}} L_2$ implies $L_1 \subseteq_{\mathcal{R}} L_2$.

Finally, linearisability is compatible with library composition. $\subseteq$ is closed under union with libraries that use disjoint stores, while $\subseteq_{\text{enc}}$ is closed under a form of sequencing that respects encapsulations.

5 Related and future work

Linearisability has been consistently used as a correctness criterion for concurrent algorithms on a variety of data structures [18], and has inspired a variety of proof methods [5]. An explicit connection between linearisability and refinement was made in [6], where it was shown that, in base-type settings, linearisability and refinement coincide. Similar results have been proved in [4, 9, 17, 3]. Our contributions are notions of linearisability that serve as correctness criteria for libraries with methods of arbitrary order and have a similar relationship to refinement. The next natural target is to investigate proof methods for establishing linearisability of higher-order concurrent libraries. The examples proved herein are only an initial step in that direction.

At the conceptual level, [6] proposed that the verification goal behind linearisability is observational refinement. In this vein, [24] utilised logical relations as a direct method for proving refinement in a higher-order concurrent setting, while [23] introduced a program logic that builds on logical relations. On the other hand, proving conformance to a history specification has been addressed in [20] by supplying history-aware interpretations to off-the-shelf Hoare logics for concurrency. Other logic-based approaches for concurrent higher-order libraries, which do not use linearisability, include Higher-Order and Impredicative Concurrent Abstract Predicates [21, 22].

Acknowledgements. We thank the authors of [3] for bringing the higher-order linearisability problem to our attention, Radha Jagadeesan and Kasper Svendsen for constructive comments, and C. Tzevelekov for help with Figure 1.
References

Higher-Order Linearisability


Quantum Markov chains are an extension of classical Markov chains which are labelled with super-operators rather than probabilities. They allow to faithfully represent quantum programs and quantum protocols. In this paper, we investigate model checking $\omega$-regular properties, a very general class of properties (including, e.g., LTL properties) of interest, against this model.

For classical Markov chains, such properties are usually checked by building the product of the model with a language automaton. Subsequent analysis is then performed on this product. When doing so, one takes into account its graph structure, and for instance performs different analyses per bottom strongly connected component (BSCC). Unfortunately, for quantum Markov chains such an approach does not work directly, because super-operators behave differently from probabilities. To overcome this problem, we transform the product quantum Markov chain into a single super-operator, which induces a decomposition of the state space (the tensor product of classical state space and the quantum one) into a family of BSCC subspaces. Interestingly, we show that this BSCC decomposition provides a solution to the issue of model checking $\omega$-regular properties for quantum Markov chains.
Introduction

Since its introduction, quantum computing has been considered a really promising technology for solving computationally complex tasks. Some of these tasks, such as factorisation and discrete logarithm computation, are the building blocks of cryptographic protocols developed to ensure security and privacy in communication. Quantum computing, by its own nature, allows for an easy solution to such tasks, so the future construction of a working quantum computer would compromise several important cryptography-based applications such as bank transactions and private communication. This has given rise to a large amount of research for providing a new class of communication protocols based on quantum mechanics so as to get back the desired properties. For instance, protocols such as super-dense coding [7], quantum coin-flipping protocol [6], and quantum key distribution protocols [6, 5] have been proposed as new building blocks for quantum cryptography.

However, as quantum mechanics is counter-intuitive, quantum protocol designers are more likely to make errors than their classical peers. This will become especially serious when more and more complicated quantum protocols can be implemented by future physical technology. Therefore, it is indispensable to develop methodologies and techniques for the verification of quantum systems.

This paper explores the possibility of applying model checking [11, 3], one of the dominant techniques for verification which has already a large number of successful industrial applications [8, 10, 21], to the verification of quantum protocols. In particular, we are interested in model checking $\omega$-regular properties, a very general class of properties subsuming those expressible by LTL formulae, against quantum Markov chains (QMCs), an extension of classical Markov chains which allow to faithfully represent quantum programs and quantum protocols. Similar to the classical case, we first take the product of the QMC and a parity automaton representing the $\omega$-regular property of interest. The model checking problem then boils down to calculating the value of the product parity quantum Markov chain (PQMC). However, we show by a counterexample that the traditional BSCC decomposition analysis used for classical model checking does not work in quantum case. To overcome this problem, we transform the product PQMC into a single super-operator on an extended Hilbert space including both the classical and quantum states. We show that due to the special structure of such an extended super-operator, the notion of BSCC subspaces for super-operators defined in [32] can be applied to tackle the problem.

1.1 Related works

The main obstacle of model checking quantum systems is that the set of all quantum states, traditionally regarded as the underlying state space of the model to be checked, is a continuum. Hence, the techniques of classical model checking, which normally work only for a finite state space, cannot be applied directly. Gay et al. [17] considered a special scenario where the initial state is a stabiliser state, and the quantum operations allowed all belong to the class of Clifford group, so that all the quantum states produced in the evolution are finitely describable. In this way, they proposed an efficient model checker [18] for certain quantum protocols, employing purely classical algorithms. Based on the same simplification, Ardestchi-Larijani et al. developed equivalence checkers for deterministic quantum protocols [1] as well as concurrent quantum protocols that behave functionally [2]. However, this approach does not work for general quantum systems. In contrast, the quantum Markov chain model adopted in this paper, which is derived from [16], is capable of describing general quantum programs and protocols, not only those in stabiliser formalism.
The state space of our quantum Markov chain is taken classical and transitions between classical states are labelled by trace-nonincreasing super-operators (thus the corresponding quantum state space is implicitly implied). In contrast, there is another notion of quantum Markov chains, which is a pair \((\mathcal{H}, \mathcal{E})\) with \(\mathcal{H}\) being a finite dimensional Hilbert space and \(\mathcal{E}\) a trace-preserving super-operator on \(\mathcal{H}\), investigated in the literature. Model checking techniques for this notion of quantum Markov chains have been extensively investigated in recent years [32, 31, 22]. These two notions of quantum Markov chains turn out to be equivalent in expressive power [22]. However, they are useful in different scenarios. The model in [32] corresponds naturally to generic quantum operations and quantum communication channels, both being popular objects of study in quantum information theory. In contrast, the quantum Markov chain model considered in this paper is more suitable for analysing quantum programs and protocols where classical states such as program counters, program variables, and measurement outcomes are naturally present.

1.2 Relevance of our work

Quantum Markov chains

The notion of quantum Markov chains studied in this paper was introduced in [16] (a similar definition was given in [19] to generalise quantum walks), which has been shown to be expressive enough to describe general quantum systems. The explicit modelling of a quantum while program and well-known quantum protocols such as teleportation, superdense coding, quantum key distribution protocol BB84, etc., can be found in [16, 15].

One of the distinct features of this model, for verification purpose, is that it provides a way to check once for all in that once a property is checked to hold, it holds for all initial quantum states. This is especially important for the verification of quantum programs. For example, for the reachability problem we calculate the accumulated super-operator, say \(\mathcal{E}\), along all valid paths. As a result, the reachability probability when the program is executed on the initial quantum state \(\rho\) is simply the trace \(\text{tr}(\mathcal{E}(\rho))\) of \(\mathcal{E}(\rho)\).

\(\omega\)-regular properties for QMCs

It has been shown in [16] how properties in quantum computation tree logic (QCTL), a quantum variant of the probabilistic CTL (PCTL), can be verified. We then provided a tool implementation [15] based on the probabilistic model checker IscasMC [20]. The applicability of this method so far was however hindered by the fact that the expressiveness of QCTL is rather limited. As the logic PCTL by which it was motivated, QCTL basically only allows to describe nested (single-step, bounded, and unbounded) reachability problems. To overcome this issue, in this paper we describe how \(\omega\)-regular properties, and in particular linear time logic (LTL) properties, can be checked on quantum Markov chains. This allows to express and analyse a wide range of relevant properties, such as repeated reachability, reachability in a restricted order, nested Until properties, or conjunctions of such properties.

Admittedly, up to now we still do not have any quantum communication protocols that have desired properties only describable in \(\omega\)-regular languages (that is also why we could not have a case study to test the effectiveness of our approach and algorithm in this paper). However, with the rapid development of quantum communication technology, especially quantum cryptographic systems, being able to check these kinds of properties for quantum Markov chains will be necessary, as they allow for instance to verify that the processes in a quantum communication protocol will repeatedly send messages, that messages are sent in
the correct order, that the key is exchanged for sure, etc., all of which cannot be expressed in QCTL.

2 Quantum Markov Chains

In this section, we recall the required notions of quantum Markov chains. For a more thorough discussion, we refer the interested reader to [24, 16].

Given a finite dimensional Hilbert space $\mathcal{H}$, let $\mathcal{L}(\mathcal{H})$ be the set of linear operators on it. Let $\mathcal{S}(\mathcal{H})$ be the set of super-operators, that is, completely positive linear operators from $\mathcal{L}(\mathcal{H})$ to $\mathcal{L}(\mathcal{H})$. In particular, we denote by $\mathcal{I}_H$ and $\mathcal{0}_H$ the identity and null super-operators in $\mathcal{S}(\mathcal{H})$, respectively. For simplicity, we abuse the notation slightly by denoting $\mathcal{E} = \{ E_i \mid i \in I \}$ if $\{ E_i \mid i \in I \}$ is a set of Kraus operators of $\mathcal{E}$; that is, $\mathcal{E}(A) = \sum_{i \in I} E_i A E_i^\dagger$ for all $A \in \mathcal{L}(\mathcal{H})$. For any $\mathcal{E}, \mathcal{F} \in \mathcal{S}(\mathcal{H})$, the composition of $\mathcal{E}$ and $\mathcal{F}$ is defined by $(\mathcal{E} \circ \mathcal{F})(A) = \mathcal{E}(\mathcal{F}(A))$. We sometimes omit the symbol $\circ$ and write $\mathcal{E}\mathcal{F}$ directly for $\mathcal{E} \circ \mathcal{F}$. A (pre-)order is defined in $\mathcal{S}(\mathcal{H})$ by setting $\mathcal{E} \lesssim \mathcal{F}$ if for any $\rho \in \mathcal{D}(\mathcal{H})$, $\text{tr}(\mathcal{E}(\rho)) \leq \text{tr}(\mathcal{F}(\rho))$. Here $\mathcal{D}(\mathcal{H})$ is the set of partial density operators in $\mathcal{L}(\mathcal{H})$, i.e., positive semidefinite operators $\rho$ with the trace $\text{tr}(\rho)$ being no larger than 1. Note that the trace of a partial density operator denotes the probability that the corresponding (normalised) quantum state is reached [28]. Intuitively, $\mathcal{E} \lesssim \mathcal{F}$ means that the success probability of performing $\mathcal{E}$ is always not greater than that of performing $\mathcal{F}$, whatever the initial state is. Let $\preceq$ be $\lesssim \cap \gtrsim$.

We denote by $\mathcal{S}^T(\mathcal{H})$ the set of trace-nonincreasing super-operators over $\mathcal{H}$; that is, $\mathcal{S}^T(\mathcal{H}) = \{ \mathcal{E} \in \mathcal{S}(\mathcal{H}) \mid \mathcal{0}_H \preceq \mathcal{E} \preceq \mathcal{I}_H \}$. Observe that $\mathcal{E} \in \mathcal{S}^T(\mathcal{H})$ if and only if for any $\rho \in \mathcal{D}(\mathcal{H})$, $\text{tr}(\mathcal{E}(\rho)) \in [0,1]$. Thus it is natural to regard the set $\mathcal{S}^T(\mathcal{H})$ as the quantum counterpart of $[0,1]$, the domain of traditional probabilities. This is exactly the key to the notion of quantum Markov chains defined in [16], that we use as our basic model.

**Definition 1** (Quantum Markov Chain). A super-operator weighted Markov chain over a Hilbert space $\mathcal{H}$, also referred to as quantum Markov chain (QMC) for simplicity, is a tuple $(\mathcal{S}, \mathcal{Q})$, where

1. $\mathcal{S}$ is a finite set of classical states;
2. $\mathcal{Q} : \mathcal{S} \times \mathcal{S} \to \mathcal{S}^T(\mathcal{H})$ is called the transition matrix where for each $s \in \mathcal{S}$, the super-operator $\sum_{s' \in \mathcal{S}} \mathcal{Q}(s, s')$ is trace-preserving, that is $\sum_{s' \in \mathcal{S}} \mathcal{Q}(s, s') \simeq \mathcal{I}_H$.

Similar to classical Markov chains, the notions of paths and measures can be defined for QMCs.

**Definition 2** (Paths and measures). Consider a QMC $\mathcal{M} = (\mathcal{S}, \mathcal{Q})$. A path $\sigma$ of $\mathcal{M}$ is a finite or infinite sequence $s_0 s_1 \ldots$ of states in $\mathcal{S}$ such that for each valid index $i \geq 1$, $\mathcal{Q}(s_{i-1}, s_i) \neq \mathcal{0}_H$. For a valid index $i$, we let $\sigma[i] \overset{\text{def}}{=} s_i$. We denote the set of finite paths as $\text{Path}^\text{fin}_\mathcal{M}$ and the set of infinite paths as $\text{Path}^\text{inf}_\mathcal{M}$. We define the cylinder set of a finite path $\sigma = s_0 s_1 \ldots s_n$ as $\text{Cyl}(\sigma) \overset{\text{def}}{=} \{ \sigma' \in \text{Path}^\text{inf}_\mathcal{M} \mid \forall i, 0 \leq i \leq n, \sigma[i] = \sigma'[i] \}$. Let $(\text{Path}^\text{inf}_\mathcal{M}, \Sigma)$ be a measurable space where $\Sigma$ is the $\sigma$-algebra generated by all the cylinder sets $\text{Cyl}(\sigma)$ where $\sigma \in \text{Path}^\text{inf}_\mathcal{M}$. For any $s \in \mathcal{S}$, we define $Q^\mathcal{M}_s : \text{Path}^\text{fin}_\mathcal{M} \to \mathcal{S}(\mathcal{H})$ as

$$Q^\mathcal{M}_s(\sigma) \overset{\text{def}}{=} \begin{cases} 0_H & s \neq s_0 \\ \mathcal{I}_H & s = s_0 \land n = 0 \\ \mathcal{Q}(s_{n-1}, s_n) \mathcal{Q}(s_{n-2}, s_{n-1}) \cdots \mathcal{Q}(s_0, s_1) & s = s_0 \land n > 0. \end{cases}$$

Then $Q^\mathcal{M}_s$ induces a (super-operator valued) measure on $(\text{Path}^\text{fin}_\mathcal{M}, \Sigma)$, denoted by $Q^\mathcal{M}_s$ as well for simplicity, by setting $Q^\mathcal{M}_s(\text{Cyl}(\sigma)) \overset{\text{def}}{=} Q^\mathcal{M}_s(\sigma)$.

From [16, Theorem 3.2], this measure is unique up to $\sim$. 
Model checking \(\omega\)-regular properties for QMCs

LTL and \(\omega\)-regular properties have been studied extensively for classical Markov chains [13, 12, 9, 3]. To compute the probability \(P(\phi)\) that a certain LTL property \(\phi\) is satisfied in a Markov chain \(M\), the classical automaton-based approach works as follows. At first, \(\phi\) is transformed into a nondeterministic Büchi automaton, which is then transformed into a deterministic automaton \(A\) with a more complex acceptance condition, such as Rabin or Parity acceptance. Such a determinisation step usually exploits a variant of Safra’s [26] determinisation construction, such as the techniques presented in [25, 27]. Afterwards, the product \(M \otimes A\) of \(M\) and \(A\) is constructed, which is a Markov chain equipped with an acceptance condition. Finally, using algorithms operating on the graph structure of the product Markov chain, the states of \(M \otimes A\) are categorised into those belonging to a \textit{bottom strongly connected component (BSCC)} and \textit{transient states}. According to the acceptance condition, each BSCC is then marked as accepting or rejecting. The probability that \(\phi\) holds in a transient state \(s\) can then be obtained by solving an equation system representing the probability that from \(s\) an accepting BSCC is reached.

This section is devoted to extending this approach to quantum Markov chains. However, the extension is not trivial: as will be shown by a counterexample in Section 3.2, while the product construction itself does not lead to any problem, its decomposition into BSCCs and transient states cannot be performed as in the classical case. Therefore, in Section 3.3, we provide an alternative approach which does not directly rely on the graph structure of the product. Specifically, we transform \(M \otimes A\) into a single super-operator, and show that the BSCC decomposition of the classical-quantum Hilbert space (the tensor product of classical state space and the quantum one) induced by this super-operator, instead of the decomposition of the classical state space alone, provides a desired solution to the model checking \(\omega\)-regular properties for quantum Markov chains.

3.1 Parity automata and parity quantum Markov chains

In order to define properties of QMCs, we consider an extension in which their states are decorated by a labelling function.

\textbf{Definition 3 (Labelled Quantum Markov Chain).} A labelled quantum Markov chain (LQMC) is a tuple \(M = (S, Q, AP, L)\), where \((S, Q)\) is a QMC and
1. \(AP\) is a finite set of atomic propositions; and
2. \(L : S \rightarrow 2^{AP}\) is a labelling function.

The notions of paths, measures, etc. for LQMCs are as in Definitions 1 and 2. We extend the labelling functions to paths by setting

\[L(s_0s_1s_2\ldots) \overset{\text{def}}{=} L(s_0)L(s_1)L(s_2)\ldots.\]

The properties we are interested in are the \(\omega\)-regular properties (which include properties definable in LTL).

\textbf{Definition 4 (\(\omega\)-regular Properties).} An \(\omega\)-regular language is a subset of \((2^{AP})^\omega\) which can be defined using an \(\omega\)-regular expression [29]. Consider an LQMC \(M = (S, Q, AP, L)\) and an \(\omega\)-regular language \(W \subseteq (2^{AP})^\omega\). We define \(Q^M_M(W) \overset{\text{def}}{=} Q^M_M(\{\sigma \in \text{Path}^M \mid L(\sigma) \in W\}).\)

We shortly restate a well-known mechanism to decide whether a word is included in a given \(\omega\)-regular language. For this purpose, an additional definition is needed.
**Definition 5** (Parity Automaton). A (deterministic) parity automaton (PA) is a tuple \(A = (A, \pi, AP, t, pri)\), where
1. \(A\) is a finite set of automaton states, and \(\pi \in A\) is the initial state,
2. \(AP\) is a finite set of atomic propositions,
3. \(t: A \times 2^{AP} \to A\) is a transition function,
4. \(pri: A \to \mathbb{N}\) is a priority function. Here \(\mathbb{N}\) denotes the set of natural numbers.

A path of \(A\) is an infinite sequence \(\sigma = a_0L_0a_1L_1\ldots \in (A \times 2^{AP})^\omega\) such that \(a_0 = \pi\) and for all \(i \geq 0\), \(t(a_i, L_i) = a_{i+1}\). We extend the priority function to paths by setting \(\text{pri}(\sigma) \overset{\text{def}}{=} \liminf_{i \to \infty} \text{pri}(a_i)\). We use \(\text{Path}^A\) to denote the set of all paths of \(A\). The language of \(A\) is defined as
\[
\mathcal{L}(A) \overset{\text{def}}{=} \{ L_0L_1\ldots \in (2^{AP})^\omega \mid \exists \sigma = a_0L_0a_1L_1\ldots \in \text{Path}^A, \text{pri}(\sigma) \text{ is even} \}.
\]

The following result is well known from the literature; see e.g. [23, 14].

**Lemma 6** (PAs represent the \(\omega\)-regular languages). A language \(W\) is \(\omega\)-regular if and only if it is the language of a PA \(A\), i.e., \(W = \mathcal{L}(A)\).

In particular this means that all properties which can be expressed in LTL can be also expressed as parity automata. Effective means to transform LTL formulas to parity automata exist in, say, [26, 25, 27].

We also need to consider QMCs with parity conditions.

**Definition 7** (Parity Quantum Markov Chain). A parity quantum Markov chain (PQMC) is a tuple \(\mathcal{M} = (S, Q, pri)\), where \((S, Q)\) is a QMC and \(\text{pri}: S \to \mathbb{N}\) is a priority function for the classical states. We define the value of \(\mathcal{M}\) in \(s \in S\) as
\[
\text{val}_s^\mathcal{M} \overset{\text{def}}{=} Q_s^\mathcal{M} \left( \{ \sigma \in \text{Path}^\mathcal{M} \mid \text{pri}(\sigma) \text{ is even} \} \right).
\]
Here again, we set \(\text{pri}(\sigma) \overset{\text{def}}{=} \liminf_{i \to \infty} \text{pri}(s_i)\) provided that \(\sigma = s_0s_1s_2\ldots\)

### 3.2 Product construction

In the following, we describe how to combine an LQMC under consideration with a PA representing the property we are concerned with.

**Definition 8** (LQMC-PA Product). The product of an LQMC \(\mathcal{M} = (S, Q, AP, L)\) and a PA \(A = (A, \pi, AP, t, pri)\) with the same set of atomic propositions is a PQMC \(\mathcal{M} \oslash A \overset{\text{def}}{=} (S', Q', pri')\) where
1. \(S' \overset{\text{def}}{=} S \times A\),
2. \(Q'((s, a), (s', a')) \overset{\text{def}}{=} Q(s, s')\) if \(t(a, L(s)) = a'\), and \(0_t\) otherwise,
3. \(pri'((s, a)) \overset{\text{def}}{=} pri(a)\).

The following lemma shows that the value of this product is trace equivalent to the super-operator corresponding to the property under consideration in the original model.

**Lemma 9.** Consider the product \(\mathcal{M}' \overset{\text{def}}{=} \mathcal{M} \oslash A = (S', Q', pri')\) of an LQMC \(\mathcal{M} = (S, Q, AP, L)\) and a PA \(A = (A, \pi, AP, t, pri)\). We have that for any \(s \in S\),
\[
Q_s^{\mathcal{M}'}(\mathcal{L}(A)) \simeq \text{val}_s^{\mathcal{M}}(\mathcal{L}(A)),
\]
**Proof.** The proof is standard.
Figure 1 Example showing that BSCC decomposition for the underlying graph does not work for model checking PQMCs.

Up to now, the model checking method works as for classical Markov chains. What would fail is the subsequent part which consists of the evaluation of the PQMC.

The idea for model checking of classical parity Markov chains is quite simple: a path of a PMC is accepted if the lowest priority occurring infinitely often is even. A strongly connected component (SCC) of a classical Markov chain is a maximal set of states \( B \) such that any two states in \( B \) can reach each other with nonzero probability. A bottom SCC (BSCC) is an SCC \( B \) in which no state in \( B \) can reach any state outside \( B \) with nonzero probability. BSCCs can be computed using only the graph structure of the Markov chain. That is, concrete probabilities are irrelevant; only the information whether the probability of going from one state to another is nonzero matters. In a classical Markov chain, starting from \( s \), the probability that \( s' \) is visited infinitely often is 1 if \( s \) and \( s' \) are in the same BSCC. The probability that a state which is not contained in any BSCC (a transient state) will be visited infinitely often is 0. Thus, model checking for PMCs can be performed as follows:

1. Identify the set of BSCCs using a graph-based algorithm, and let \( ACC = \emptyset \).
2. For each BSCC \( B \), check whether the lowest priority occurring on a state of \( B \) is even. If yes, add \( B \) to \( ACC \), \( ACC \leftarrow ACC \cup B \).
3. For any state \( s \), if \( s \in ACC \), then \( val_s^M = 1 \). Otherwise, \( val_s^M \) is the probability that \( s \) reaches any state in \( ACC \). That is, if \( s \) is a state of a BSCC \( B \not\subseteq ACC \), then \( val_s^M = 0 \); and values of transient states can be computed by solving a linear equation system.

Note that a PQMC also has a set of classical states, and the transition super-operators also induce an underlying graph over these states. Thus a natural question is: can we define the notion of BSCCs in terms of the underlying graph structure for a PQMC, just as in the classical case, and employ the above technique to calculate its value? Unfortunately, this idea does not work, as the following example shows. A similar example illustrating this difficulty was also given in [22].

Example 10. Consider the two parity Markov models in Figure 1. On the left is a classical one with \( 0 < p < 1 \), while the right is a quantum one with \( \mathcal{E}_0, \mathcal{E}_1 \neq 0_H \) and \( \mathcal{E}_0 + \mathcal{E}_1 \cong I_H \). Obviously, both models have the same classical state space, and have exactly the same underlying graph. Thus they have the same set of BSCCs, if we would define BSCCs for PQMCs according to the underlying graphs. However, we will see that this BSCC technique does not help in the evaluation of PQMCs.

In the classical model, \( s_0 \) is a transient state which will eventually reach the only BSCC \( \{s_1, s_2\} \). Thus, the priority with which \( s_0 \) is labelled is irrelevant. From any state of the BSCC, the probability that both states are visited infinitely often is 1. Thus, the probability that from either state the lowest priority 0 is reached infinitely often is 1, and thus the value of the parity Markov chain is also 1.

In contrast, in the quantum model, we assume \( \mathcal{E}_0 \equiv \{|0\rangle\langle 0|\} \) and \( \mathcal{E}_1 \equiv \{|1\rangle\langle 1|\} \). Note that for \( i \in \{0, 1\} \) it holds \( \mathcal{E}_i \mathcal{E}_i = \mathcal{E}_i \) and \( \mathcal{E}_i \mathcal{E}_{i-1} = 0_H \). It is easy to check that if we start from \( s_0 \), the infinite path \((s_0)^\omega\), with the corresponding nonzero super-operator \( \lim_{n \to \infty} \mathcal{E}_1^n = \{|1\rangle\langle 1|\} \),
never leaves to the set \( \{s_1, s_2\} \). Thus \( s_0 \) should not be considered as a transient state at all. Furthermore, as the priority of \((s_0)^\omega\) is 0, this path also contributes to the value of PQMC. On the other hand, if we start from \( s_1 \), there are two infinite paths with nonzero super-operator, namely \((s_1)^\omega\) with the corresponding super-operator \( \{\langle 1|1 \rangle\} \) and priority 0, and \((s_1s_2)^\omega\) with the corresponding super-operator \( \{\langle 0|0 \rangle\} \) and priority 0. Thus, the value of the PQMC in state \( s_1 \) is \( \{\langle 0|0 \rangle\} + \{\langle 1|1 \rangle\} \approx \mathcal{I}_H \). However, if we start from \( s_2 \) we have \((s_2)^\omega\) with the corresponding super-operator \( \{\langle 1|1 \rangle\} \) and priority 1, and \((s_2s_1)^\omega\) with the corresponding super-operator \( \{\langle 0|0 \rangle\} \) and priority 0. Thus, the value in \( s_2 \) is \( \{\langle 0|0 \rangle\} \), different from the one in \( s_1 \).

Thus, algorithms based on BSCC decomposition of the underlying graph do not work for PQMCs: neither are BSCCs reached with certainty, nor do all states of a BSCC have the same value. In addition, the value of a BSCC state might be equivalent to neither \( 0_H \) nor \( \mathcal{I}_H \).

### 3.3 Computing PQMC values

We have seen from Example 10 that the notion of BSCC defined for the underlying graph over classical states does not help in evaluation of PQMCs. In this subsection, we show that, rather surprisingly, by encoding the behavior of \( \mathcal{M} \) into a single super-operator acting on the extended Hilbert space which is the tensor product of the classical state space and the quantum one, the notion of BSCC subspaces for super-operators\(^1\) defined in [32] can be used to compute PQMC values.

We first recall some definitions from [32]. For any \( \rho \in \mathcal{D}(\mathcal{H}) \), the support \( \text{supp}(\rho) \) is defined to be the space spanned by the eigenvectors of \( \rho \) with non-zero eigenvalues. Let \( \{X_k\} \) be a family of subspaces of \( \mathcal{H} \). The join of \( \{X_k\} \) is defined as \( \bigvee_k X_k = \text{span}(\bigcup_k X_k) \).

Let \( \mathcal{E} \) be a super-operator acting on \( \mathcal{H} \) with \( \text{dim}(\mathcal{H}) = d \). A subspace \( X \) of \( \mathcal{H} \) is said to be invariant for \( \mathcal{E} \) if \( \mathcal{E}(X) \subseteq X \), and it is a BSCC of \( \mathcal{E} \) if \( \mathcal{R}(|\psi\rangle\langle\psi|) = X \) for any pure state \( |\psi\rangle \in X \), where for any \( \rho \in \mathcal{D}(\mathcal{H}) \),

\[
\mathcal{R}(\rho) = \bigvee_{i=0}^{\infty} \text{supp}(\mathcal{E}^i(\rho))
\]

is the reachable subspace of \( \mathcal{E} \) starting in \( \rho \). Apparently, a BSCC is also an invariant subspace. Finally, \( X \) is called transient if \( \lim_{k \to \infty} \text{tr}(P_X \mathcal{E}^k(\rho)) = 0 \) for any \( \rho \in \mathcal{D}(\mathcal{H}) \), where \( P_X \) is the projection onto \( X \).

Let \( \mathcal{M} = (S, Q, \text{pri}) \) be a PQMC on \( \mathcal{H} \) with \( Q(s,t) = \{ E_i^{s,t} \mid i \in I^{s,t} \} \). Following [22], we define a super-operator

\[
\mathcal{E}_\mathcal{M} \overset{\text{def}}{=} \{ |t\rangle\langle s| \otimes E_i^{s,t} \mid s, t \in S, i \in I^{s,t} \}
\]

acting on the Hilbert space \( \mathcal{H}_c \otimes \mathcal{H} \), where \( \mathcal{H}_c \) is a \( |S| \)-dimensional Hilbert space with an orthonormal basis \( \{|s\} \mid s \in S \} \). To see how \( \mathcal{E}_\mathcal{M} \) encodes the behavior of \( \mathcal{M} \), let for each

---

\(^1\) We choose not to use the terminology quantum Markov chain as in [32], to avoid confusion with the notion of quantum Markov chain defined in this paper. Interestingly, although it has been observed that these two notions of quantum Markov chains have the same expressiveness power [22], this is the first time techniques from one model find applications in the other.
Let \( s \in S \) that \( O^s \defeq \{(l, i) \mid t \in S, i \in I^{s,t}\} \), and \( M^s \defeq \{ E_i^{s,t} \mid (t, i) \in O^s \} \). Note that for any \( \rho \in \mathcal{D}(\mathcal{H}) \),

\[
\text{tr} \left( \sum_{(t, i) \in O^s} E_i^{s,t} \rho (E_i^{s,t})^\dagger \right) = \text{tr} \left( \sum_{t \in S} Q(s, t)(\rho) \right) = \text{tr}(\rho)
\]

where the second equality comes from the fact that \( \sum_{t \in S} Q(s, t) \simeq \mathcal{I}_\mathcal{H} \) (see Definition 1). That is, \( M^s \) is actually a quantum measurement with the outcome set \( O^s \). Furthermore, for any \( \sigma \in \mathcal{D}(\mathcal{H}_c) \) and \( \rho \in \mathcal{D}(\mathcal{H}) \), we calculate that

\[
E_M(\sigma \otimes \rho) = \sum_{s, t \in S, i \in I^{s,t}} |t\rangle \langle s| \otimes E_i^{s,t} \rho (E_i^{s,t})^\dagger = \sum_{s \in S} \langle s| \sigma \rangle \sum_{(t, i) \in O^s} |t\rangle \langle t| \otimes E_i^{s,t} \rho (E_i^{s,t})^\dagger.
\]

Thus the behavior of \( E_M \) can be described as the following steps, which exactly captures the intended meaning of \( M \):

1. a projective measurement \( M \defeq \{|s\rangle \langle s| : s \in S\} \) is performed on the classical system \( \mathcal{H}_c \) to determine the current classical state;
2. if the measurement outcome of \( M \) is \( s \), then the quantum measurement \( M^s \) is performed on the quantum system \( \mathcal{H} \);
3. the classical state is set to be \( |t\rangle \langle t| \) if \( (t, i) \), for any \( i \), is observed in \( M^s \).

The following lemma shows that for super-operators derived from PQMCs, the classical and quantum systems will remain separable (disentangled) during the evolution, provided that the initial state is in a product form.

\( \blacktriangleright \text{Lemma 11.} \text{ Let } M = (S, Q, \text{pri}) \text{ be a PQMC on } \mathcal{H}, s \in S, k \in \mathbb{N}, \text{ and } \rho \in \mathcal{D}(\mathcal{H}). \text{ Then for any } n \geq 0, \mathcal{E}_M^n(|s\rangle \langle s| \otimes \rho) \text{ is block diagonal according to the classical states. Specifically,}
\]

\[
\mathcal{E}_M^n(|s\rangle \langle s| \otimes \rho) = \sum_{t \in S} |t\rangle \langle t| \otimes Q^n(s, t)(\rho).
\]

2. Let \( R^k_s = Q^n_M(|\sigma \in \text{Path}^M | \text{pri}(\sigma) = k \}) \). Then for any \( n \geq 0 \),

\[
\text{tr}(R^k_s(\rho)) = \sum_{t \in S} \text{tr}(R^k_t(Q^n(s, t)(\rho))).
\]

\( \text{Proof.} \text{ Statement 1 is easy by induction. For Statement 2, note that for any } n \geq 0,
\]

\[
R^k_s = Q^n_M(|\sigma \in \text{Path}^M(s) | \lim_{i \geq 0} \text{pri}(\sigma[i]) = k \})
\]

\[
\simeq \sum_{t \in S} Q^n_M(|\sigma \in \text{Path}^M(t) | \lim_{i \geq 0} \text{pri}(\sigma[i]) = k \}) \circ Q^n(s, t)
\]

\[
= \sum_{t \in S} R^k_t \circ Q^n(s, t).
\]

Then the result follows. \( \blacktriangleleft \)
can be spanned by pure states of the form $|s\rangle|\psi\rangle$ where $s \in S$ and $|\psi\rangle \in \mathcal{H}$. For a BSCC $B$ of $\mathcal{E}_M$, let

$$C(B) \overset{\text{def}}{=} \{ s \in S \mid |s\rangle|\psi\rangle \in B \text{ for some } |\psi\rangle \in \mathcal{H} \}$$

be the set of classical states supported in $B$.

Exploiting the classical-quantum separation (Lemma 11) of super-operators derived from PQMCs, we are going to show some nice properties of their BSCC decomposition, which are key to our discussion in this paper. First, we prove that two BSCCs $X$ and $Y$ are orthogonal, denoted $X \perp Y$, unless they have the same set of support classical states.

**Lemma 12.** Let $\mathcal{M}$ be a PQMC. For any two BSCCs $X$ and $Y$ of $\mathcal{E}_M$, if $C(X) \neq C(Y)$ then $X \perp Y$.

**Proof.** Suppose $C(X) \neq C(Y)$, and, without loss of generality, let $s \in C(Y) \setminus C(X)$. Let $\rho_X$ and $\rho_Y$ be the fixed point states corresponding to $X$ and $Y$, respectively. Since $\mathcal{E}_M(\rho_X + \rho_Y) = \rho_X + \rho_Y$, we know that $(\rho_X + \rho_Y)/2$ is a fixed point state corresponding to $Z \overset{\text{def}}{=} X \lor Y$. Thus $Z$ can be decomposed into the direct sum of some orthogonal BSCCs: $Z = X \oplus Z_1 \oplus \cdots \oplus Z_n$.

We claim that $n = 1$. Otherwise, for any $i$, $\dim(Z_i) < \dim(Y)$ (because $\sum_i \dim(Z_i) + \dim(X) = \dim(Z) = \dim(X) + \dim(Y)$), and thus $Y \perp Z_i$ by [32, Lemma 2]. This means $Y = X$, a contradiction. Now let $|s\rangle|\psi\rangle \in Y$. Since $s \notin C(X)$, we have $|s\rangle|\psi\rangle \perp X$, and thus $|s\rangle|\psi\rangle \in Z_1$. On the other hand, since both $Y$ and $Z_1$ are BSCCs, $Y = Z_1 = R(|s\rangle\langle s|\otimes |\psi\rangle\langle \psi|)$. Thus $X \perp Y$.

Given $k \in \mathbb{N}$, let $\text{BSCC}_k$ be the span of all BSCCs of $\mathcal{E}_M$ with the minimal priority being $k$; that is,

$$\text{BSCC}_k = \sqrt{\{ B \text{ is a BSCC of } \mathcal{E}_M : \min\{\text{pri}(s) \mid s \in C(B)\} = k \}}.$$

Similarly, let $\text{BSCC}_{k-}$ and $\text{BSCC}_{k+}$ be the spans of all BSCCs with the minimal priority being less than and larger than $k$, respectively. Then by Lemma 12, $\text{BSCC}_k$, $\text{BSCC}_{k-}$, and $\text{BSCC}_{k+}$ are pairwise orthogonal. From [32], the state space $\mathcal{H}_c \otimes \mathcal{H}$ can be decomposed uniquely into

$$\mathcal{H} = T \oplus \text{BSCC}_k \oplus \text{BSCC}_{k-} \oplus \text{BSCC}_{k+},$$

where $T$ is the maximum transient subspace of $\mathcal{E}_M$. In the following, we denote by $P_T$, $P_k$, $P_{k-}$ and $P_{k+}$ the projections onto $T$, $\text{BSCC}_k$, $\text{BSCC}_{k-}$ and $\text{BSCC}_{k+}$, respectively. Then $P_T + P_k + P_{k-} + P_{k+} = I_{\mathcal{H}_c \otimes \mathcal{H}}$, the identity operator on $\mathcal{H}_c \otimes \mathcal{H}$.

The following lemma is crucial for our purpose. Note that $\text{tr}(R^t(\rho))$ denotes the probability that $k$ is the lowest priority infinitely often reachable from the initial state $|t\rangle\langle t| \otimes \rho$. This lemma essentially says that such a probability will be $1$ if starting from $\text{BSCC}_k$ (provided that $\text{tr}(\rho) = 1$; otherwise, the probability is $\text{tr}(\rho)$), and it will be $0$ if starting from either $\text{BSCC}_{k-}$ or $\text{BSCC}_{k+}$. Thus $\text{BSCC}_k$ for each $k$ acts like the standard BSCCs in classical Markov chains.

**Lemma 13.** For any $t \in S$ and $\rho \in \mathcal{D}(\mathcal{H})$,

1. if $\text{supp}(|t\rangle\langle t| \otimes \rho) \subseteq \text{BSCC}_k$, then $\text{tr}(R^t(\rho)) = \text{tr}(\rho)$;
2. if $\text{supp}(|t\rangle\langle t| \otimes \rho) \subseteq \text{BSCC}_{k-}$, then $\text{tr}(R^t(\rho)) = 0$; and
3. if $\text{supp}(|t\rangle\langle t| \otimes \rho) \subseteq \text{BSCC}_{k+}$, then $\text{tr}(R^t(\rho)) = 0$.

With the above lemmas, we are now ready to prove the main theorem of this section.
Theorem 14. Given a PQMC $\mathcal{M} = (S, Q, \rho_{\text{pri}})$ and $k \in \mathbb{N}$, for any $s \in S$ and $\rho \in \mathcal{D}(\mathcal{H})$,
\[\text{tr}(R_k^s(\rho)) = \text{tr}(P_k \mathcal{E}_s^\infty(|s\rangle\langle s| \otimes \rho))\]
where $\mathcal{E}_s^\infty = \lim_{N \to \infty} \frac{1}{N} \sum_{n=1}^{N} \mathcal{E}_s^n$.

Proof. First we lift the set of super-operators $R_k^s$ on $\mathcal{H}$ to $\mathcal{H}_c \otimes \mathcal{H}$ by defining $\tilde{R}_k^s \triangleq \sum_{t \in S} \mathcal{E}_t \otimes R_k^s$ where $\mathcal{E}_t = \{|t\rangle\langle t|\}$. Then Lemma 11 says that for any $n \geq 0$, $\text{tr}(R_k^s(\rho)) = \text{tr}(\tilde{R}_k^s(\mathcal{E}_s^n(\rho)))$ where $\rho_s \triangleq |s\rangle\langle s| \otimes \rho$. Thus we have
\[\text{tr}(R_k^s(\rho)) = \text{tr}\left(\tilde{R}_k^s \left(\frac{1}{N} \sum_{n=1}^{N} \mathcal{E}_s^n(\rho_s)\right)\right)\]
for any $N \geq 1$, and so $\text{tr}(R_k^s(\rho)) = \text{tr}(\tilde{R}_k^s \mathcal{E}_s^\infty(\rho_s))$ by letting $N$ tend to infinity.

On the other side, note that $\mathcal{E}_s^\infty \triangleq \mathcal{E}_s^\infty(\rho_s)$ is a fixed point state of $\mathcal{E}_s$. Then by Lemma 12 and [4, Theorem 6], $P_T \rho_s^\infty P_T = 0$, and $\rho_s^\infty$ is block diagonal with respect to $\text{BSCC}_k$, $\text{BSCC}_k^\perp$, and $\text{BSCC}_{k+}$; that is, $\rho_s^\infty = P_k \rho_s^\infty P_k + P_{k-} \rho_s^\infty P_{k-} + P_{k+} \rho_s^\infty P_{k+}$. Thus from Lemma 13,
\[
\text{tr}(R_k^s(\rho)) = \text{tr}(\tilde{R}_k^s(P_k \rho_s^\infty P_k)) + \text{tr}(\tilde{R}_k^s(P_{k-} \rho_s^\infty P_{k-})) + \text{tr}(\tilde{R}_k^s(P_{k+} \rho_s^\infty P_{k+}))
\]
\[= \text{tr}(P_k \rho_s^\infty).\]

The following corollary, which is direct from Theorem 14, provides us a neat way to represent the value of a PQMC at a given state using certain super-operators without resorting to quantum states.

Corollary 15. Let $\mathcal{M} = (S, Q, \rho_{\text{pri}})$ be a PQMC. Then for any $s \in S$,
\[\text{val}_s^\mathcal{M} \simeq \text{tr}_c \circ \mathcal{P}_{\text{even}} \circ \mathcal{E}_s^\infty \circ \mathcal{E}_s\]
where $\text{tr}_c$ is the partial trace super-operator such that $\text{tr}_c(|s\rangle\langle t| \otimes \rho) = \langle t| s \rangle \cdot \rho$, $\mathcal{P}_{\text{even}} = \sum_{k \in \text{pri}(S) | k \text{ is even}} P_k$ where $P_k = \{P_k\}$ is the projection super-operator onto $\text{BSCC}_k$, and $\mathcal{E}_s(\rho) = |s\rangle\langle s| \otimes \rho$.

Note that in the above corollary, we do not calculate the value $\text{val}_s^\mathcal{M}$ directly. Instead, only a super-operator which is trace equivalent to $\text{val}_s^\mathcal{M}$ is obtained. Note that from [16, Theorem 3.2], the super-operator valued measure for a QMC (thus PQMC) is well-defined only up to the trace equivalence $\simeq$. In other words, it does not make sense to talk about the exact super-operator associated with a measurable set of paths; only the equivalence class determined by $\simeq$ is meaningful. Fortunately, this is sufficient for our purpose, as in practice we are interested in the probability of satisfying a certain $\omega$-property, starting from an initial quantum state, which depends only on the equivalence class that the super-operator like $\text{val}_s^\mathcal{M}$ is in.

Example 16 (Example 10 revisited). Let $\mathcal{M}$ be the PQMC depicted on the right of Fig. 1 where $\mathcal{E}_0 \triangleq \{|0\rangle\langle 0|\}$ and $\mathcal{E}_1 \triangleq \{|1\rangle\langle 1|\}$. Then the super-operator encoding $\mathcal{M}$ is
\[
\mathcal{E}_\mathcal{M} = |s_1\rangle\langle s_0| \otimes \mathcal{E}_0 + |s_0\rangle\langle s_0| \otimes \mathcal{E}_1
\]
\[+ |s_1\rangle\langle s_1| \otimes \mathcal{E}_1 + |s_2\rangle\langle s_1| \otimes \mathcal{E}_0
\]
\[+ |s_1\rangle\langle s_2| \otimes \mathcal{E}_0 + |s_2\rangle\langle s_2| \otimes \mathcal{E}_1,
\]
the maximal transient space of $\mathcal{E}_M$ is $T \overset{\text{def}}{=} \text{span}\{|s_0\rangle\langle 0\rangle\}$, and the BSCCs are

\[ B_1 \overset{\text{def}}{=} \text{span}\{|s_0\rangle\langle 1\rangle\}, \quad B_2 \overset{\text{def}}{=} \text{span}\{|s_1\rangle\langle 1\rangle\}, \quad B_3 \overset{\text{def}}{=} \text{span}\{|s_1\rangle\langle 0\rangle, |s_2\rangle\langle 0\rangle\}, \quad B_4 \overset{\text{def}}{=} \text{span}\{|s_2\rangle\langle 1\rangle\}. \]

Thus $\text{BSCC}_0 = \sqrt{\{B_1, B_2, B_3\}}$, and $P_0 = |s_0\rangle\langle s_0| \otimes |1\rangle\langle 1| + |s_1\rangle\langle s_1| \otimes I + |s_2\rangle\langle s_2| \otimes |0\rangle\langle 0|$. Furthermore, we calculate that for any $n \geq 1$, $\mathcal{E}_M^{2n-1} = \mathcal{F}_0 \otimes \mathcal{E}_0 + \mathcal{F} \otimes \mathcal{E}_1$ and $\mathcal{E}_M^\infty = \mathcal{F}_1 \otimes \mathcal{E}_0 + \mathcal{F} \otimes \mathcal{E}_1$ where $\mathcal{F}_0 \overset{\text{def}}{=} \{|s_1\rangle\langle s_0|, |s_2\rangle\langle s_1|, |s_1\rangle\langle s_2|\}$, $\mathcal{F}_1 \overset{\text{def}}{=} \{|s_2\rangle\langle s_0|, |s_1\rangle\langle s_1|, |s_2\rangle\langle s_2|\}$, and $\mathcal{F} \overset{\text{def}}{=} \{|s_0\rangle\langle s_0|, |s_1\rangle\langle s_1|, |s_2\rangle\langle s_2|\}$. Thus $\mathcal{E}_M^\infty = \mathcal{F}_0 + \mathcal{F}_1 \otimes \mathcal{E}_0 + \mathcal{F} \otimes \mathcal{E}_1$, and

\[ P_0 \circ \mathcal{E}_M^\infty = \mathcal{F}_0 + \mathcal{F}_1 \otimes \mathcal{E}_0 + (P_{s_0} + P_{s_1}) \otimes \mathcal{E}_1. \]

Note that $\mathcal{E}_s = \{|s\rangle \otimes I\}$ and $\text{tr}_c = \{|s_i\rangle \otimes I \mid i = 0, 1, 2\}$. It follows that

\[ \text{val}_{s}^M \approx \text{tr}_c \circ P_0 \circ \mathcal{E}_M^\infty \circ \mathcal{E}_s = \begin{cases} \mathcal{E}_0 + \mathcal{E}_1 \approx \mathcal{I}_H & \text{if } s = s_0 \lor s = s_1 \\ \mathcal{E}_0 & \text{if } s = s_2, \end{cases} \]

coinciding with the informal discussion given in Example 10.

4 The algorithm

In this section, we propose an algorithm to compute the values of a PQMC. First, we introduce some notations. For a super-operator $\mathcal{E} = \{E_i \mid i \in I\}$ acting on Hilbert space $\mathcal{H}$, let $M_{E} = \sum_{i,j} E_i \otimes E_j^*$ be its matrix representation which is a linear operator on $\mathcal{H} \otimes \mathcal{H}$. Here the complex conjugate $E_j^*$ is taken according to an orthonormal basis of $\mathcal{H}$. It is easy to check that $M_{E}$ is independent of the choices of orthonormal basis and Kraus operators $E_i$ of $\mathcal{E}$. Let $M_{E} = KJK^{-1}$ be the Jordan decomposition of $M_{E}$ where $J = \bigoplus_k J_{\lambda_k}$ and $J_{\lambda_k}$ is a Jordan block corresponding to the eigenvalue $\lambda_k$. Define

\[ J^\infty = \bigoplus_{\{k\mid \lambda_k = 1\}} J_{\lambda_k} \quad (2) \]

and $M_{E}^\infty = KJ^\infty K^{-1}$. Then from [30, Proposition 6.3], $M_{E}^\infty$ is the matrix representation of $\mathcal{E}_E^\infty$.

\[ \textbf{Theorem 17.} \text{Given a PQMC } M = (S, Q, \text{pri}) \text{ on } \mathcal{H} \text{ and a classical state } s \in S, \text{Algorithm 1 computes the matrix representation of a super-operator which is trace equivalent to } \text{val}_{s}^M \text{ in time } O(n^4d^8), \text{where } n = |S| \text{ and } d = \text{dim}(\mathcal{H}). \]

\[ \textbf{Proof.} \text{Note that for any super-operators } \mathcal{E} \text{ and } \mathcal{F}, \text{the matrix representation of } \mathcal{E} \circ \mathcal{F} \text{ is the product of matrix representations of } \mathcal{E} \text{ and } \mathcal{F}; \text{that is, } M_{E} \circ M_{F} = M_{E \circ F}. \text{Then the correctness of Algorithm 1 follows from Corollary 15. The Procedure GetBSCC which, given a super-operator } \mathcal{E} \text{ and an invariant subspace of } \mathcal{E}, \text{outputs a complete set of orthogonal BSCCs in that subspace is a revised version of the procedure Decompose from [32].} \]

\[ \text{Note that } \text{dim}(\mathcal{H} \otimes \mathcal{H}) = nd, \text{and the matrix representation of } \mathcal{E}_M \text{ has size } n^2d^2 \times n^2d^2. \text{The complexity of Algorithm 1 can be estimated as follows.} \]

\begin{enumerate}
\item The time complexity of computing the matrix representation $M$ of $\mathcal{E}_M$ is $\sum_{s,t \in S} m_{s,t} d^4 = O(n^2d^8)$, where $m_{s,t} \overset{\text{def}}{=} |I^{s,t}| \leq d^2$ is the number of Kraus operators in $Q(s,t)$.
\item Note that the time complexity of Jordan decomposition is $O(m^4)$ for an $m \times m$ matrix. The computation of matrix representation $M^\infty$ of $\mathcal{E}_M^\infty$ takes time $O(n^8d^8)$.\end{enumerate}
Algorithm 1: Compute the values of a PQMC

\textbf{input} : A PQMC, $\mathcal{M} = (S, Q, \text{pri})$ on $\mathcal{H}$ and a classical state $s \in S$.
\textbf{output} : (Matrix representation of) a super-operator that is trace equivalent to $\text{val}_s^\mathcal{M}$.
begin
(* Compute the matrix representations of $\mathcal{E}_s$, $\text{tr}_c$, and $\mathcal{E}_M$ *)
\begin{align*}
E_s & \leftarrow |s\rangle \otimes I_\mathcal{H}; M_s \leftarrow E_s \otimes E_s^*; \\
M_c & \leftarrow 0;
end \text{for } t \in S \text{ do}
\begin{align*}
E & \leftarrow \langle t | \otimes I_\mathcal{H}; M_c & \leftarrow M_c + E \otimes E^*;
\end{align*}
\text{end}
\begin{align*}
M & \leftarrow 0;
\text{for } t, t' \in S \text{ and } i \in I^{t, t'} \text{ do}
\begin{align*}
E & \leftarrow \langle t' | \otimes E_{t'}^*; M & \leftarrow M + E \otimes E^*; \\
\text{(* Compute the matrix representation of $\mathcal{P}_{\text{even}}$ *)}
\end{align*}
\text{end}
\begin{align*}
(K, J) & \leftarrow \text{Jordan decomposition of } M; \\
M^\infty & \leftarrow K J K^{-1}; \\
\text{(* Compute the matrix representation of $\mathcal{E}_M$ *)}
\end{align*}
\begin{align*}
M_{\text{even}} & \leftarrow 0; I_c & \leftarrow \sum_{t \in S} |t\rangle \langle t|; \\
B & \leftarrow \text{GetBSCC}(M, I_c \otimes I_\mathcal{H}); \\
E P & \leftarrow \{ \text{pri}(t) \mid t \in S \land \text{pri}(t) \text{ is even} \}; \\
\text{for } k \in E P \text{ do}
\begin{align*}
P_k & \leftarrow 0; \\
\text{for } B \in B \text{ with } k = \min \{ \text{pri}(t) \mid t \in C(B) \} \text{ do}
\begin{align*}
P_k & \leftarrow P_k + P_B \text{ where } P_B \text{ is the projector onto } B; \\
\end{align*}
\text{end}
M_{\text{even}} & \leftarrow M_{\text{even}} + P_k \otimes P_k; \\
\end{align*}
\text{end}
\text{return } M_c \times M_{\text{even}} \times M^\infty \times M_s; \text{ (* $\times$ denotes normal matrix multiplication *)}
end
\end{align*}
\begin{align*}
3. \text{ For the Procedure GetBSCC}(M, I_\mathcal{H}) \text{ where } I_\mathcal{H} \overset{\text{def}}{=} \mathcal{H}_c \otimes \mathcal{H}, \text{ the most time-consuming step is to compute the null space of the matrix } I_\mathcal{H} \otimes I_\mathcal{H} - M. \text{ This can be done by Guassian elimination with complexity being } O((n^2 d^2)^3) = O(n^6 d^6). \text{ Note that each recursive call of the procedure decreases the dimension of the subspace by at least one. The complexity of computing GetBSCC}(M, I_\mathcal{H}) \text{ is } O(n^7 d^7). \end{align*}

At first glance, the time complexity $O(n^6 d^6)$ of Algorithm 1 looks very high. However, note that a typical super-operator on a $d$-dimensional Hilbert space has up to $d^2$ Kraus operators each of them is a $d \times d$ complex matrix. Thus the input size $K$ of a PQMC $\mathcal{M} = (S, Q, \text{pri})$ is actually of order $O(n^2 d^4)$ with $n = |S|$. Thus the time complexity of Algorithm 1 is indeed $O(K^4)$.

Note that the decomposition of $M$ (the matrix representation of $\mathcal{E}_M$) into Jordan blocks in Algorithm 1 is quite expensive. Therefore, for a practical implementation, an approximate approach might be preferable. From $\mathcal{E}_M^\infty = \lim_{N \to \infty} \frac{1}{N} \sum_{n=1}^N \mathcal{E}_M^n$ (cf. Theorem 14) we can derive its matrix representation $M^\infty = \lim_{N \to \infty} M_N$ where $M_N \overset{\text{def}}{=} \frac{1}{N} \sum_{n=1}^N M^n$. We then
Model Checking $\omega$-regular Properties for Quantum Markov Chains

The procedure `GetBSCC(M, P)` is as follows:

**Input**: The matrix representation $M$ of a super-operator $\mathcal{E}$ acting on $\mathcal{H} = \mathcal{H}_c \otimes \mathcal{H}$, and a projector $P$ to some invariant subspace $\mathcal{H}^I \subseteq \mathcal{H}$ of $\mathcal{E}$.

**Output**: A complete set of orthogonal BSCCs of $\mathcal{E}$ in $\mathcal{H}$.

```
Procedure GetBSCC(M, P)

begin

    input : The matrix representation $M$ of a super-operator $\mathcal{E}$ acting on $\mathcal{H} = \mathcal{H}_c \otimes \mathcal{H}$, and a projector $P$ to some invariant subspace $\mathcal{H}^I \subseteq \mathcal{H}$ of $\mathcal{E}$.
    output: A complete set of orthogonal BSCCs of $\mathcal{E}$ in $\mathcal{H}$.

    begin

        $\{ |\psi_i \rangle | i \in I \}$ ← an orthonormal basis of $\mathcal{H}$;
        $\mathcal{X}$ ← a basis of $\{|x\rangle \in \mathcal{H} \otimes \mathcal{H} | M|x\rangle = |x\rangle \}$;
        $F$ ← $\emptyset$;

        for $|x\rangle \in \mathcal{X}$ do
            $\begin{cases}
                X ← \sum_{i \in I} (I_R \otimes \langle \psi_i |) \cdot |x\rangle \langle \psi_i |; \\
                (* The matrix $X$ corresponds to $|x\rangle$ in that $X = \sum_{i,j \in I} x_{i,j} |\psi_i \rangle \langle \psi_j |$ iff $|x\rangle = \sum_{i,j \in I} x_{i,j} |\psi_i \rangle \langle \psi_j |$ *)
            \end{cases}$

            $X_R ← (X + X^\dagger)/2$;
            $X_I ← (X - X^\dagger)/2i$ ;

            $\begin{cases}
                P^+_R ←$ the projector onto eigenspace of $X_R$ with positive eigenvalues;
                P^+_I ←$ the projector onto eigenspace of $X_I$ with positive eigenvalues;
                X^+_R = P^+_R X_R P^+_R \otimes X^+_R = X^+_R - X_R;
                X^+_I = P^+_I X_I P^+_I \otimes X^+_I = X^+_I - X_I;
            \end{cases}$

            $(*$ All of them are positive semidefinite, and $X = X^+_R - X_R + i(X^+_I - X_I) * )$

        for $Y \in \{ X^+_R, X^-_R, X^+_I, X^-_I \}$ do
            if $|F| = 1$ then
                return $\{ \text{supp}(Y) \}$;
            else
                $Y_1, Y_2 ←$ two arbitrary different elements of $F$;
                $P^+ ←$ the projector onto eigenspace of $Y_1 - Y_2$ with positive eigenvalues;
                $P^- ← P - P^+$;
                $M^+ ← (P^+ \otimes I_R) M (I_R \otimes P^+)$;
                $M^- ← (P^- \otimes I_R) M (I_R \otimes P^-)$;
                return $\text{GetBSCC}(M^+, P^+) \cup \text{GetBSCC}(M^-, P^-)$;
            \end{cases}$

        end

    end

end
```

compute $M_0, M_1, M_2, \ldots$ until we have reached an $N$ for which $\|M_N - M_{N-1}\|_{\text{max}} < \varepsilon$ for a predefined precision $\varepsilon$, so as to obtain an approximation of $M^\infty$. Note that $M_N$ can be computed using a dynamic programming approach by means of the equality

$$M_N = \begin{cases}
    M & \text{if } N = 1, \\
    \frac{1}{N} ((N-1)M_{N-1} + M^N) & \text{if } N > 1.
\end{cases}$$

In stochastic model checking, such value iteration based approaches are commonly used, and in [15], we have successfully applied a similar method for model checking QCTL Until formulas.
5 Conclusion

In this paper, we have investigated model checking $\omega$-regular and in particular LTL properties against super-operator weighted quantum Markov chains, which can be used to faithfully model a practically relevant class of quantum processes. As future work, we would like to implement our model checking algorithm in IscasMC [20] and apply it on case studies from the area of quantum communication protocols and evaluate the actual performance of our approach.

References

Model Checking $\omega$-regular Properties for Quantum Markov Chains


25 Nir Piterman. From nondeterministic Büchi and Streett automata to deterministic parity automata. JLMCS, 3(3:5), 2007.


Uniform Sampling for Networks of Automata*†

Nicolas Basset¹, Jean Mairesse², and Michèle Soria³

1 Université libre de Bruxelles, Brussels, Belgium
nicholas.basset@ulb.ac.be
2 Sorbonne Universités, UPMC Univ Paris 06, CNRS, LIP6, 4 place Jussieu,
75252 Paris Cedex 05, France
jean.mairesse@lip6.fr
3 Sorbonne Universités, UPMC Univ Paris 06, CNRS, LIP6, 4 place Jussieu,
75252 Paris Cedex 05, France
michele.soria@lip6.fr

Abstract

We call network of automata a family of partially synchronised automata, i.e. a family of deterministic automata which are synchronised via shared letters, and evolve independently otherwise. We address the problem of uniform random sampling of words recognised by a network of automata. To that purpose, we define the reduced automaton of the model, which involves only the product of the synchronised part of the component automata. We provide uniform sampling algorithms which are polynomial with respect to the size of the reduced automaton, greatly improving on the best known algorithms. Our sampling algorithms rely on combinatorial and probabilistic methods and are of three different types: exact, Boltzmann and Parry sampling.

1998 ACM Subject Classification F.1.1 Models of Computation, F.1.2 Modes of Computation, G.2.1 Combinatorics, G.3 Probability and Statistics

Keywords and phrases Partially synchronised automata, uniform sampling, recursive method, Boltzmann sampling, Parry measure

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2017.36

1 Introduction

Automata are ubiquitous in computer science in general, and in verification in particular, since they provide a good abstraction of the behaviour of sequential systems. Networks of automata are a meaningful model of concurrent systems where a family of component automata are synchronised via shared letters, and otherwise evolve independently. They appear under various names in the literature (see e.g. [3, 18] and references therein). An example of a network of automata is depicted in Figure 1. A challenging task when dealing with such concurrent model is to avoid the state space explosion due to an explicit construction of the product model.

In the context of either performance evaluation or model-checking of concurrent systems, it is often impossible to perform a formal or exhaustive analysis of the huge number of possible trajectories (recognised words in the context of automata). To cope with the issue, a possibility is to perform either simulation or Monte Carlo model checking, that is, to

* An extended version of this article is available in [4].
† This work was partially supported by the Fondation Simone et Cino Del Duca. N. Basset was partially supported by the ARC project “Non-Zero Sum Game Graphs: Applications to Reactive Synthesis and Beyond” funded by Fédération Wallonie-Bruxelles.

© Nicolas Basset, Jean Mairesse, and Michèle Soria; licensed under Creative Commons License CC-BY
28th International Conference on Concurrency Theory (CONCUR 2017).
Editors: Roland Meyer and Uwe Nestmann; Article No. 36; pp. 36:1–36:16
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany
Figure 1 Top: A network of three DFAs with shared alphabet $\text{Synch} = \{\alpha, \beta, \gamma\}$. The dotted transitions labelled with $\gamma$ (top) have been added without changing the product, see Remark 2. Bottom: the product. Unreachable states $113, 123, 213, 223, 312, 322$ are not represented. The states occurring just after a synchronisation are $112, 311, 313, 323$.

concentrate on a sample of the trajectories drawn at random (see [13, 20]). As noted in [9, 20], the classical sampling methods applied in this context use a random walk evolving on the product automaton, the most natural one being the isotropic random walk that chooses the next transition uniformly at random among available transitions. For instance, for the example of Figure 1, when the state $112$ is visited by the random walk then each one of the transitions $\alpha, a$ and $b$ is chosen with probability $\frac{1}{3}$. The isotropic sampling seems very natural at first sight but, unfortunately, there are simple examples (exhibited in [9, 20]) for which this sampling tends to concentrate most of the probability on a very small fraction of the trajectories. Moreover, isotropic sampling is useless to answer quantitative question such as: 'With which confidence can we claim than more than $60\%$ of the trajectories of length $1000$ visit a target state?'. To answer such questions, it is necessary to use a uniform sampling that is to choose all equal length trajectories with the same probability. A thorough argumentation why uniform sampling should be preferred to isotropic sampling is given in [9, 20] for both Monte Carlo model checking and random model based testing. Despite these pioneering works, designing efficient uniform samplers for network of automata remains a challenge that we address in the present paper.

Before detailing our work, we need to distinguish between different notions of uniform sampling. A fixed length uniform sampler is a random algorithm taking as input a positive integer $n$ and returning as output a word of length $n$ with uniform probability. A Boltzmann sampler is a random algorithm returning a word of random length with the property that
two words of the same length are equiprobable. A Parry sampler is a random method to generate infinite words in a "uniform" way. It requires the rigorous definition of the notion of "uniform probability measure" on infinite words. Both Boltzmann and Parry samplings are relevant in Monte Carlo model checking. On the one hand, if some variability on the length of the sampled word is allowed, then Boltzmann sampling is useful and efficient, and the expected length of the sampled word can be chosen by tuning a parameter. On the other hand, if some variability with respect to uniformity is allowed, then Parry sampling is of interest: with Parry, we obtain an exact uniform sampling for infinite words, but finite words can also be sampled, with approximate uniformity and an additional important feature: the sampling procedure is dynamic, that is, we can re-use a sampled word of length $n$ as the prefix of a sampled word of length $n+k$.

The straightforward method of building explicitly the product automaton (the automaton defined as the direct product of the component automata), and apply to it a standard uniform sampling algorithm for automata (e.g. [5, 19]), is not efficient since the product automaton has a size which is exponential with respect to the size of the components. In fact, the whole challenge is to avoid constructing explicitly the product automaton. To be efficient, the approach has to be compositional: use uniform samplers for component automata and combine them in a clever way.

We now review previous literature related to the problem. Consider first the case with no shared letters, that is, no synchronisation between the component automata. In this case, the language of the network of automata is simply the shuffle of the languages of the component automata. The shuffle product has been widely studied from a formal language theory viewpoint (see [21] and references therein). We are not aware of any paper dealing explicitly with uniform sampling in this precise context. The most relevant article is [8] which provides a Boltzmann sampler for extended linear expressions with shuffle. Applying this algorithm in our context is not straightforward since we do not know a priori if there exists a polynomial method to transform a family of automata into an extended linear expression with shuffle. The direct method, which consists in shuffling the regular expressions obtained by transforming each component automaton separately, is not efficient. Indeed, the passage from automata to regular languages is known to increase the size of the description in an exponential way in the worst case (see e.g. [2]). In [9] the authors identify (amongst other contributions) the challenge of uniform sampling for network of DFAs. Our work is partly inspired by ideas of this paper like using shuffle of languages between synchronisations. However, there are several problems unsolved there that make their sampling useful only under very stringent conditions:

(i) there must be at most one shared letter in the network of automata and if there is one, it must appears exactly once in every component automaton;
(ii) there must be very few synchronisation in the sampled word (the algorithm having an exponential complexity wrt. the number of occurrences of the shared letter in the sampled word);
(iii) the sampling algorithm for the shuffle of languages can only be used if the words that are shuffled are long and belong to strongly connected automata.

In fact, we show that the third item is not a real problem. We provide exact uniform samplers for the shuffle of languages that are built and that run in polynomial time without any restriction on the length of the words shuffled nor on the topology of the component automata. By contrast, when all the letters are synchronised, difficulties are unavoidable in the worst-case. Indeed, the problem of uniform sampling is more difficult than checking the language emptiness of the product of automata which is a PSPACE complete problem. An
option to address this issue would be to identify a subclass for which the problem would be tractable in polynomial time like in the first item above. Here we avoid such a restriction by adopting a fixed-parameter-tractable approach: our algorithms are designed for the whole class of networks of automata but run in polynomial time when a well suited parameter is fixed. This parameter controls the size of the 'synchronised part' of the product automaton. We thus take advantage of the fact that the synchronised part of the product may be small, and decompose the network into pieces coming from the synchronised part and others coming from the 'shuffle' of non-synchronised parts, leading to efficient sampling algorithms that use the product only when necessary. To that purpose, we introduce the notion of reduced automaton. Each component automaton of the network is transformed into a simplified automaton involving only the shared letters, and a remaining part. The reduced automaton is the product of the simplified automata, hence the exponential blow-up only concerns the synchronised part of the automata, and the size of the reduced automaton is a parameter that reflects the intrinsic synchronisation complexity of the problem. Our basic idea is to decompose the language recognized by the network of automata in terms of union and concatenation of shuffles of languages corresponding to the component automata. Relying on this decomposition, we then use a compositional approach to derive uniform samplers.

We now sum up our results. We propose a fixed length uniform sampler for arbitrary network of automata running in linear time in the length of the sampled word, and in polynomial time in the size of the reduced automaton. In particular we provide solutions to the problems of [9] listed above: there is no restriction on the number of shared letters; the sampling is exactly uniform; the complexity is drastically improved wrt. the number of occurrences of shared letters in the word. In the case of no shared letters, we design a fixed length uniform sampler running in linear time wrt. the length of the sampled words and in polynomial time wrt. the sum of the sizes of the component automata. This is a key ingredient used in the case where synchronisations are present. In addition to this, we provide the first Boltzmann and Parry samplers for networks of automata.

We conclude this introduction with few related works. Uniform sampling for related but different models of concurrent systems are considered in [1, 7]. Approximate uniform sampling of valuations of SAT-formulas are given in [16] (see also references therein).

2 Preliminaries: Monolithic Uniform Sampling for a DFA

In this section, we present the three types of uniform sampling for a single automaton. All the algorithms are classical and run in polynomial time. They will be used as building blocks in the compositional approach of the following sections.

Let us begin with some basics on automata. A finite state automaton (FSA) is a tuple $A = (Q, \Sigma, \iota, F, \Delta)$ where $Q$ is the set of states, $\Sigma$ is the alphabet of actions, $\iota$ is the initial state, $F$ is the set of final states and $\Delta \subseteq Q \times \Sigma \times Q$ is the set of transitions. A path of length $n \in \mathbb{N}$ from $s$ to $t$ labelled by $w = a_1 \cdots a_n \in \Sigma^n$ is a sequence of transitions $(s_i, a_i, t_i)_{i \in [n]}$ such that $s_1 = s$, $t_n = t$, for $i < n$, $t_i = s_{i+1}$ (where here and below, for $k \in \mathbb{N}$ we let $[k] = \{1, \ldots, k\}$). We write $s \xrightarrow{w} t$ if there is a path from $s$ to $t$ labelled by $w$. A word $w$ is recognised by the automaton $A$ if there is a final state $t \in F$ such that $s \xrightarrow{w} t$. The language recognised by $A$, denoted by $L(A)$, is the set of words recognised by $A$. We denote by $L_s$ the language of words starting from $s$, that is recognised by the FSA $(Q, \Sigma, s, F, \Delta)$. The size of $A$, denoted by $|A|$, is the number of states and transitions. When $\Delta$ is functional, that is $\forall s \in Q, \forall a \in \Sigma, |\{t \mid (s, a, t) \in \Delta\}| \leq 1$, the FSA is called a deterministic finite state automaton (DFA). In automata theory, trimming is a standard operation which consists in
deleting useless states of a FSA (a state is useful if it is accessible from \( \iota \) and co-accessible from a final state). We assume without loss of generality that all automata we consider are trimmed.

For \( s \in Q \), the languages \( L_s \) can be characterised by the following language equations:

\[
L_s = \bigcup_{(s,a,t) \in \Delta} aL_t \quad (\cup \{ \varepsilon \} \text{ if } s \in F)
\]  

(1)

In the remainder of the section, we consider a DFA \( A \) of language \( L \). We are going to present three methods to sample a word from \( L \), the three being built on a common recursive scheme based on Eq. (1): randomly choose the first transition \((s,a,t)\), output the letter \( a \) and then repeat recursively from \( t \). These methods are sequential: letters are randomly chosen one after the other in the order in which they appear in the output word.

### 2.1 Cardinalities and fixed length uniform sampling

Recall that a fixed length uniform sampler is a random algorithm that takes as input a positive integer \( n \) and outputs a word of \( L \) of length \( n \) such that every word has the same probability to be output.

The general recursive method for uniform sampling of [12] applies in this context. The idea is to transfer recursive equations on cardinalities into recursive samplers. The equations on cardinalities are obtained directly from Eq. (1). Denoting by \( l_{s,n} \) the number of words of length \( n \) in \( L_s \), we have

\[
l_{s,n} = \sum_{(s,a,t) \in \Delta} l_{t,n-1} \quad \text{if } n > 0 \text{ and } l_{s,0} = 1_{s \in F}
\]  

(2)

A fixed length uniform sampler is then obtained as follows: choose the first transition \((s,a,t)\) with probability \( l_{t,n-1}/l_{s,n} \), output \( a \) and recursively repeat for the \( n-1 \) remaining letters, starting in state \( t \).

Note that a \(|Q| \times n\) table with the coefficients \((l_{s,k})_{s \in Q, k \in [n]}\) can be computed in time\(^1\) \( O(n|A|) \) using Eq. (2). Cardinalities can also be expressed in terms of the power of the adjacency matrix of \( A \), that is, the matrix \( A = (A_{st})_{s,t \in Q} \) with \( A_{st} = |\{(a) \mid (s,a,t) \in \Delta\}| \). Indeed, for all \( s \in Q, n \in \mathbb{N} \), we have: \( l_{s,n} = \sum_{t \in F} A_{st}^n \).

The drawback of the above method is that the transition probabilities \((l_{t,n-1}/l_{s,n})\) depend on \( n \) so that the cardinalities should be computed and stored up to the length of the word to be generated. In the next two sampling methods on the other hand, the transition probabilities do not depend on \( n \).

### 2.2 Generating functions and Boltzmann sampling

The general Boltzmann sampling of [10] applies in this context. Whereas the fixed length sampler was based on recursive equations on cardinalities, the Boltzmann sampler is based on recursive equations on generating functions.

Let us recall some basics on generating functions associated to languages. The ordinary generating function (OGF) of a language \( L \) is \( L(z) = \sum_{m \in \mathbb{N}} l_m z^m \), where \( l_m \) is the number of words of length \( m \) in the language, and the exponential generating function (EGF) is \( \hat{L}(z) = \sum_{m \in \mathbb{N}} l_m z^m / m! \). Generating functions can be seen either as formal power series

\[^1\] Here and in the rest of the paper, the complexity is given in terms of arithmetic complexity.
or as functions of the complex variable $z$. The convergence radius of $L(z)$ is $\tau(L) = \inf\{ |z| ; L(z) \text{ is not defined} \}$.

The equations on languages (1) transfer to equations on generating functions:

$$L_s(z) = z \sum_{(s,a,t) \in \Delta} L_t(z) + 1_{s \in F}$$

Define the column vector $L(z) = (L_s(z))_{s \in Q}$ where $L_s(z)$ is the OGF of $L_s$. Recall that $A$ is the adjacency matrix. For $z < \tau(L)$, we have: $L(z) = (I - zA)^{-1}e_F$, where $e_F$ is the column vector defined by $(e_F)_s = 1$ if $s \in F$ and $(e_F)_s = 0$ otherwise. The convergence radius $\tau(L)$ is characterised by $2^2 \tau(L) = \inf\{ |z| ; (I - zA)^{-1} \text{ is defined} \}$.

An (ordinary) Boltzmann sampler draws a word $w$ with probability distribution $z^{|w|}/L(z)$ ($z < \tau(L)$ being a parameter to be chosen), hence the size is not fixed but the distribution is uniform when conditioned on a given size. Based on Eq. (3), a Boltzmann sampler for $L_s$ of parameter $z$ can be recursively defined: with probability $1_{s \in F}/L_s(z)$, no transition is chosen and the random generation stops; otherwise the transition $(s,a,t)$ is chosen with probability $zL_t(z)/L_s(z)$, the letter $a$ is output, and the sampler is called recursively from $t$ to output the remainder of the word.

Similarly, an exponential Boltzmann sampler draws words with probability distribution $z^{|w|}/(|w|!\hat{L}(z))$. We give in [4] the construction of such an exponential Boltzmann sampler.

### 2.3 Perron-Frobenius Theorem and Parry sampling

In this section, we need to assume that the DFA under consideration is strongly connected\(^3\).

Parry sampling is intuitively the limit case of Boltzmann sampling where $z = \tau(L)$ so that the probability to stop the generation is null and the output word is infinite. The formal definition is based on the Perron-Frobenius Theorem. This theorem gives fundamental properties on the spectral theory of non-negative matrices. We state it in the context of strongly connected automata. We refer the reader to [15].

**Theorem 1 (Perron-Frobenius stated for automata).** Consider a strongly connected DFA and its adjacency matrix $A$. The following holds:

(i) The spectral radius $\rho(A)$, that is, the maximal modulus of all eigenvalues of $A$, is itself an eigenvalue. It satisfies $\rho(A) = 1/\tau(L)$.

(ii) The eigenvalue $\rho(A)$ is simple, its unique (up to a multiplicative constant) eigenvector $\mathbf{v}$ has positive coefficients, it is called the Perron vector. There is no other eigenvector with only non-negative coefficients.

(iii) If $0 \leq \lambda' \leq \lambda$ then $\rho(A') \leq \rho(A)$, and $\rho(A') = \rho(A)$ only if $A' = A$.

A Parry sampler for $L_s$ is an algorithm that produces an infinite random word $w$ such that for any finite word $w$ such that $s \xrightarrow{w} t$, the probability that $w$ begins by $u$ is $\mathbf{v}_t/(\rho^n\mathbf{v}_s)$, where $\mathbf{v}$ is the Perron vector. A Parry sampler can be recursively defined by choosing the first transition $(s,a,t)$ with probability $\mathbf{v}_t/(\rho\mathbf{v}_s)$ and repeating recursively from $t$.

A Parry sampler does not give exact uniform sampling on words of length $n$. However it gets closer and closer to being uniform as $n$ gets larger. Hence it can be used as an approximate uniform sampler for large words of a given length.

\(^2\) Here, we use the assumption that the DFA is trimmed, otherwise one can construct examples where a part of the DFA is useless for the language definition, but decreases the value of $\inf\{ |z| ; (I - zA)^{-1} \text{ is defined} \}$ which is in that case strictly smaller than $\tau(L)$.

\(^3\) There is a path between each couple of states in the automaton.
3 Network of automata and the reduced automaton

A network of automata is composed of a family of DFA that are synchronised on shared letters and evolve independently otherwise. The associated reduced automaton is a product automaton taking into account only the synchronised part of the components. In this section, we formally define these notions and provide equations on languages associated to states and transitions of the reduced automaton, together with a system of equations satisfied by their generating functions.

3.1 Network of automata

Consider a family of \( K \) DFAs \( A^{(i)} = (Q^{(i)}, \Sigma^{(i)}, \epsilon^{(i)}, F^{(i)}, \Delta^{(i)}) \) for \( i \in [K] \). The alphabets \( \Sigma^{(i)} \) are not assumed to be disjoint. The associated product automaton is the DFA defined by \( A^{(1)} \times \cdots \times A^{(K)} = (Q, \Sigma, \epsilon, F, \Delta) \) with \( Q = Q^{(1)} \times \cdots \times Q^{(K)} \), \( \Sigma = \Sigma^{(1)} \cup \cdots \cup \Sigma^{(K)} \); \( \epsilon = (\epsilon^{(1)}, \ldots, \epsilon^{(K)}) \); \( F = F^{(1)} \times \cdots \times F^{(K)} \) and \( (s, a, t) \in \Delta \) if and only if: \( \forall i \in [K] \) s.t. \( a \in \Sigma^{(i)} \), \( (s^{(i)}, a, t^{(i)}) \in \Delta^{(i)} \); and \( \forall i \in [K] \) s.t. \( a \not\in \Sigma^{(i)} \), \( s^{(i)} = t^{(i)} \).

An example is given in Figure 1. We call network of automata a family of DFAs evolving together according to the above rules of the product automaton.

Denote by Synch the set of letters shared by several automata of the network (we use Greek letters for the elements of Synch).

\[ \text{Remark 2. Given } (A^{(i)})_{i \in [K]}, \text{ we define for every } i \in [K], \text{ the FSA } B^{(i)} \text{ obtained from } A^{(i)} \text{ by adding in every state a self-loop labelled by every } \alpha \in \text{Synch} \setminus \Sigma^{(i)}. \text{ Observe that we have: } A^{(1)} \times \cdots \times A^{(K)} = B^{(1)} \times \cdots \times B^{(K)}. \text{ See an example in Figure 1.} \]

Accordingly, in the following, we assume without loss of generality that every letter in Synch belongs to every alphabet.

The special case where Synch = \( \emptyset \) will be of crucial importance. It involves the shuffle of languages defined just below. The shuffle product of two words \( w^{(1)} \) and \( w^{(2)} \) on disjoint alphabets is the finite language, denoted by \( w^{(1)} \sqcup w^{(2)} \), containing all the possible interleavings of \( w^{(1)} \) and \( w^{(2)} \), e.g. \( ab \sqcup cd = \{ abed, acbd, acdb, cdab \} \). The shuffle product of two languages \( L^{(1)} \) and \( L^{(2)} \) on disjoint alphabets is

\[ L^{(1)} \sqcup L^{(2)} = \bigcup_{(w^{(1)}, w^{(2)}) \in L^{(1)} \times L^{(2)}} w^{(1)} \sqcup w^{(2)}. \]

The shuffle product is associative and we denote by \( L^{(1)} \sqcup \cdots \sqcup L^{(K)} \) the shuffle product of \( K \) languages \( L^{(1)}, \ldots, L^{(K)} \). The language associated to a network of automata such that Synch = \( \emptyset \) is the shuffle of the component languages:

\[ L(A^{(1)} \times \cdots \times A^{(K)}) = L(A^{(1)}) \sqcup \cdots \sqcup L(A^{(K)}). \]  

(4)

In the dual special case where Synch = \( \Sigma \), the language associated to the network of automata is the intersection of the component languages:

\[ L(A^{(1)} \times \cdots \times A^{(K)}) = L(A^{(1)}) \cap \cdots \cap L(A^{(K)}). \]

(5)

3.2 Reduced automaton

We now introduce the reduced automaton of a network of automata, a product automaton which only involves the synchronised letters. All the algorithms presented in this paper will require to compute \( A_{\text{red}} \), whereas the product automaton \( A \) is never explicitly computed.
We define the reduced automaton of a DFA $A = (Q, \Sigma, \iota, F, \Delta)$ as the FSA $A_{\text{red}} = (Q_{\text{red}}, \Sigma_{\text{red}}, \iota_{\text{red}}, Q_{\text{red}}, \Delta_{\text{red}})$ such that $\Sigma_{\text{red}} = \text{Synch}$ and $Q_{\text{red}} \subseteq Q$ is the set of states reached from the initial state $\iota_{\text{red}} = \iota$ through the transition relation $\Delta_{\text{red}} \subseteq Q_{\text{red}} \times \Sigma_{\text{red}} \times Q_{\text{red}}$ defined by $\delta = (s, \alpha, t) \in \Delta_{\text{red}}$ if and only if there exists a word $u \in (\Sigma \setminus \text{Synch})^*$ such that $s \xrightarrow{u \alpha} t$. The set of final states $F_{\text{red}} = Q_{\text{red}}$ is not relevant since we are only interested in the states and transitions of $A_{\text{red}}$, and not in its language. See Figure 2 for an example.

**Proposition 3.** Given a DFA $A$, the FSA $A_{\text{red}}$ can be computed by replacing every label not in $\text{Synch}$ by $\varepsilon$ and doing an $\varepsilon$-transitions removal.

The removal of $\varepsilon$-transitions can be achieved in time $O(|Q|^2 + |Q||\Delta|)$, as shown for instance in [17].

The next proposition enables us to construct compositionally the reduced automaton associated to a network of automata, based on the reduced automata of each of the component automata computed monolithically as in Proposition 3.

**Proposition 4.** Consider a network of automata $(A^{(i)})_{i \in [K]}$ and $A = A^{(1)} \times \cdots \times A^{(K)}$ its associated product. It holds that $A_{\text{red}} = A_{\text{red}}^{(1)} \times \cdots \times A_{\text{red}}^{(K)}$.

The problem of uniform random generation of words in the language of a product automaton requires first to check its emptiness which is PSPACE-complete in the sum of the size of the component automata [14] already for the special case with all letters synchronised (see (5)).

Our sampling algorithm described in Section 4.2 below relies on pre-computations that are polynomial in the size $|A_{\text{red}}| = |Q_{\text{red}}| + |\Delta_{\text{red}}|$ of the reduced automaton $A_{\text{red}}$. More precisely these problems are fixed parameter tractable when the size $|A_{\text{red}}|$ is considered a parameter, namely their complexity is of the form $O(|A_{\text{red}}| \cdot p(n, \sum_{i=1}^{K} |A^{(i)}|))$ where $p$ is a polynomial, $n$ is the length of words to be generated and $\sum_{i=1}^{K} |A^{(i)}|$ is the size of the network. Note that when there are only shared letters the reduced automaton has exactly the same states and transitions as the product automaton and hence can be of an exponential size. However, in the general case, when a fair proportion of letters are not shared we expect the reduced automaton to be of far smaller size than the product automaton. We give in Proposition 5 below upper-bounds on $|A_{\text{red}}|$ that involve parameters easier to compute (in polynomial time wrt. $|N|$) that we define now.

Given a letter $\alpha \in \text{Synch}$ and an automaton $A$, we denote by $d(\alpha, A)$ the cardinalities of the set of states of $A$ that are destinations of edges labelled by $\alpha$, formally $d(\alpha, A) = |\{q \mid \exists p \in Q, (p, \alpha, q) \in \Delta\}|$. Further, let $d(A) = \max_{\alpha} d(\alpha, A)$. The size of the reduced automata, which is an important parameter in the complexity of our algorithms, can be bounded:
Proposition 5. Given a network of automata \((A^{(i)})_{i \in [K]}\), the size of the reduced automaton \(A_{\text{red}}\) satisfies \(|Q_{\text{red}}| \leq d(A_{\text{red}}) \cdot |\text{Synch}| + 1\) and \(|\Delta_{\text{red}}| \leq |Q_{\text{red}}| \cdot |\text{Synch}|\) with
\[
d(A_{\text{red}}) \leq \max_{\alpha \in \text{Synch}} \prod_{i=1}^{K} d(\alpha, A_{\text{red}}^{(i)}) \leq \left( \max_{\alpha, i \leq K} d(\alpha, A^{(i)}) \right)^K.
\]

Remark 6. If every transition labelled by the same action goes to a dedicated state, then the proposition above gives \(d(A_{\text{red}}) \leq 1\), thus \(|Q_{\text{red}}| \leq |\text{Synch}| + 1\) and \(|\Delta_{\text{red}}| \leq |Q_{\text{red}}| \cdot |\text{Synch}|\). The particular case assumed in [9] where for each synchronised action \(\alpha\) there is only one occurrence of \(\alpha\) per component automaton yields further \(|\Delta_{\text{red}}| = |\text{Synch}|\).

3.3 Equations on languages and generating functions

Let \(A\) be a DFA and \(A_{\text{red}}\) its reduced automaton. For \(s \in Q_{\text{red}}\), let \(\tilde{L}_s\) be the language recognised by the DFA \(A_s\) obtained from \(A\) by removing transitions labelled by actions in \(\text{Synch}\) and by changing the initial state to \(s\). For \(\delta = (s, \alpha, t) \in \Delta_{\text{red}}\), let \(\tilde{L}_\delta\) be the language recognised by the DFA \(A_\delta\) obtained from \(A\) by removing transitions labelled by actions in \(\text{Synch}\) and by changing the initial state to \(s\) and the final states to \(\{q \mid (q, \alpha, t) \in \Delta\}\).

The following theorem constitutes the core of our study: it gives a decomposition of the language recognised by a network of automata only in terms of union and concatenation of shuffles of languages corresponding to the component automata. All our sampling algorithms for networks of automata in Section 4.2 will rely on this decomposition.

Theorem 7. Let \((A^{(i)})_{i \in [K]}\) be a network of automata and \(A_{\text{red}}\) its associated reduced automaton. Using the above notations, for every \(s \in Q_{\text{red}}\), it holds that:
\[
L_s = \tilde{L}_s \cup \bigcup_{\delta = (s, \alpha, t) \in \Delta_{\text{red}}} \tilde{L}_\delta \alpha L_t, \quad \text{with} \quad \tilde{L}_s = \bigcup_{i=1}^{K} \tilde{L}_{s,\alpha,t}^{(i)} \quad \text{and} \quad \tilde{L}_\delta = \bigcup_{i=1}^{K} \tilde{L}_{\delta,\alpha,t}^{(i)}
\]
The language operations (product, union, shuffle) involved in this equation are unambiguous.\(^4\)

The language operations translate into identities on cardinalities and generating functions: union and concatenation of languages yield sum and product of OGF, and shuffle of languages yield product of EGF (see e.g. [6, 22, 11]). To do such translations, it is crucial that the language operations are unambiguous; that is why we consider only network of deterministic automata. Denote by \(L_s(z), \tilde{L}_s(z), \tilde{L}_\delta(z)\) the OGF of languages \(L_s, \tilde{L}_s, \tilde{L}_\delta\).

Proposition 8. Let \((A^{(i)})_{i \in [K]}\) be a network of automata and \(A_{\text{red}}\) its associated reduced automaton. For every \(s \in Q_{\text{red}}\), the following equation holds on OGF:
\[
L_s(z) = \tilde{L}_s(z) + z \sum_{\delta = (s, \alpha, t) \in \Delta_{\text{red}}} \tilde{L}_\delta(z)L_t(z).
\]
And the EGF of \(\tilde{L}_s\) (resp. \(\tilde{L}_\delta\)) is the product of the EGF of \(L_{s,\alpha,t}^{(i)}\) (resp. \(L_{\delta,\alpha,t}^{(i)}\)).

Let \(M(z)\) be the \(Q_{\text{red}} \times Q_{\text{red}}\) matrix such that \([M(z)]_{s,t} = \sum_{\delta = (s, \alpha, t) \in \Delta_{\text{red}}} \tilde{L}_\delta(z)\). Then Proposition 8 can be written in terms of vectors and matrices of OGF as follows: \(L(z) = \tilde{L}(z) + z M(z) L(z)\). Based on this, we obtain the characterisation of \(v(L)\) and \(L(z)\) in the next theorem. The important point is that it depends only on the matrix \(M\) of size \(Q_{\text{red}}\) and not on the exponentially big adjacency matrix of the product automaton.

Let \(v(L) = \min_{s \in Q_{\text{red}}} v(\tilde{L}_s)\) and \(v^*(M) = \inf \{|z| \mid (I - z M(z))^{-1}\}\) is defined.\(^4\)

\(^4\) Recall that a language operation is said to be unambiguous if every word of the resulting language can be obtained in a unique way by composing different words from the operands.
Proposition 9. The convergence radius \( r(L) \) and the vector of generating function \( L(z) \) for \( z < r(L) \) are characterised as follows:

\[
r(L) = \min \left( r(L), r^\ast(M) \right) \quad \text{and} \quad L(z) = (I - zM(z))^{-1} \tilde{L}(z).
\]

4 Uniform sampling for a network of automata

Consider a network of automata. Relying on the reduced automaton, we adapt the sampling methods developed for a unique automaton, and recalled in Section 2, in order to design sampling algorithms for the network of automata. The generic method is as follows:

1. choose whether a synchronisation will occur. In the case of no synchronisation, generate a word in the shuffle \( \hat{\mathcal{L}}_s = \bigcup_{i=1}^K \hat{\mathcal{L}}^{(i)} \). In the case of synchronisation:
2. choose a transition \( \delta = (s, \alpha, t) \in \Delta_{\text{red}} \), (this gives the next synchronisation \( \alpha \)),
3. choose a word without synchronisation \( w \in \hat{\mathcal{L}}_\delta = \bigcup_{i=1}^K \hat{\mathcal{L}}^{(i)} \),
4. write \( w\alpha \) on the output tape, and repeat from \( t \) to generate the rest of the word.

This method is derived from the equations on languages in Theorem 7. It will be consistently applied in all three methods of sampling. It first requires to be able to generate words in the shuffle of languages, and second to compute the right probabilities in order to make the choices. In all cases, we assume that we have algorithms dealing with a single automaton, see Section 2 (we call them monolithic), and we adapt and combine them.

4.1 Pure Shuffle

We first study the case of no synchronisation between the component DFAs, that is \( \text{Synch} = \emptyset \). The language of the network of automata is the shuffle of the languages of the component automata (see (4)). We present the three sampling methods. They share the common idea of generating words for the component automata and then choosing a word in their shuffle.

4.1.1 Fixed length uniform sampler

For two languages, the cardinalities of the shuffle are easy to compute: if \( \mathcal{L} = \mathcal{L}_1 \sqcup \mathcal{L}_2 \), then \( l_n = \sum_{m=0}^n \binom{n}{m} l_{1,m} l_{2,n-m} \). The generalisation to \( K \) languages is more complicated:

\[
l_n = \sum_{n^{(1)} \times \cdots \times n^{(K)}} \frac{n!}{n^{(1)}! \cdots n^{(K)}!} l^{(1)}_{n^{(1)}} \cdots l^{(K)}_{n^{(K)}} \quad \text{with} \quad n^{(1)} + \cdots + n^{(K)} = n.
\]

This is not satisfactory since it involves exponentially many terms. The difficulty was already noted in [9] where a solution was proposed under restricted assumptions (in particular the strong connectivity of the DFA). Here we propose another way to bypass the difficulty which is always valid.

The idea is to decompose the shuffle in a recursive manner. We define \( \mathcal{L}^{(\leq 0)} = \{ \varepsilon \} \) and for \( 1 \leq i \leq K \), \( \mathcal{L}^{(i)} = \mathcal{L}^{(\leq i-1)} \sqcup \mathcal{L}^{(i)} \). Then \( \mathcal{L}^{(1)} \sqcup \cdots \sqcup \mathcal{L}^{(K)} = \mathcal{L}^{(\leq K)} \), and the corresponding cardinalities are recursively computed efficiently in Algorithm 1.

Algorithm 1 uses the monolithic routine \text{Mono-Cardinalities}(A, n)\), that outputs all the cardinalities \( (l_m)_{0 \leq m \leq n} \), in time \( O(n|\mathcal{A}|) \), using Eq. (2). The cardinalities associated to \( \mathcal{L}(A) \) corresponds to \( s = t \), that is, \( l_m = l_{t,m} \).

Lemma 10. The cardinalities \( (l_m^{(i)})_{0 \leq m \leq n, 0 \leq i \leq K} \) associated to the languages \( (\mathcal{L}^{(i)})_{0 \leq i \leq K} \) can be computed with Algorithm 1 in time \( O(Kn \log n + n \sum_{i=1}^K |\mathcal{A}^{(i)}|) \).
**Algorithm 1** Shuffle-Card($(A^{(i)}))_{i \in [K]}$, $n$)  
(precomputation for Shuffle-Unif).

**Require:** $K$ DFAs $A^{(i)}$ and a natural integer $n \in \mathbb{N}$.
**Ensure:** compute and store $(l_m^{(i)})_{0 \leq m \leq n, 1 \leq i \leq K}$ and $(l_m^{(\leq i)})_{0 \leq m \leq n, 1 \leq i \leq K}$.

1. for $i = 1$ to $K$
2. $(l_m^{(i)})_{0 \leq m \leq n} \leftarrow$ Mono-Cardinalities($A^{(i)}$, $n$);
3. compute $\sum_{m=0}^{n} l_m^{(i)} z^m / m! = \left( \sum_{m=0}^{n} l_m^{(\leq i-1)} z^m / m! \right) \left( \sum_{m=0}^{n} l_m^{(i)} z^m / m! \right)$ mod $z^{n+1}$

**Algorithm 2** Uniform sampler Shuffle-Unif($(A^{(i)}))_{i \in [K]}$, $n$).

**Require:** $K$ DFAs $A^{(i)}$, $n \in \mathbb{N}$, and cardinalities $(l_m^{(i)})_{0 \leq m \leq n, 1 \leq i \leq K}$, $(l_m^{(\leq i)})_{0 \leq m \leq n, 1 \leq i \leq K}$.
**Ensure:** return a word in $L^{(1)} \cup \cdots \cup L^{(K)}$ of length $n$ uniformly at random.

1. $N \leftarrow n$;
2. for $i = K$ down to 1 do
3. choose $n^{(i)}$ with probability $n^{(i)} \rightarrow \binom{N}{n^{(i)}} l_{N-n^{(i)}} / n^{(i)}$;
4. $w^{(i)} \leftarrow$ Mono-Unif($A^{(i)}$, $n^{(i)}$); // (use e.g. algorithm of [5] or [19])
5. $N \leftarrow N - n^{(i)}$;
6. return Shuffle-Words($(w^{(i)})_{i \in [K]}$). // (use e.g. algorithm of [9])

Algorithm 2 repeatedly chooses, according to the precomputed cardinalities, the length of the words to be generated in each language $L^{(i)}$. It generates such words $w_i$ using a monolithic sampling algorithm on DFA Mono-Unif (see [5] or [19]), and then returns a random word in the shuffle language of the $w_i$ by using a function Shuffle-Words that chooses uniformly at random a word in $w^{(1)} \cup \cdots \cup w^{(K)}$ in linear time (see [9]).

Sampling with Algorithm 2 is no more difficult than doing independently uniform sampling for the component automata. In the theorem below, $\text{Mono-Unif}(A^{(i)}, n)$ denotes the complexity of a monolithic algorithm for the uniform sampling of words of length $n$ in the $i$th component automaton.

**Theorem 11.** Algorithm 2 returns a word in $L^{(1)} \cup \cdots \cup L^{(K)}$ of length $n$ uniformly at random in time complexity $O(\sum_{i=1}^{K} |\text{Mono-Unif}(A^{(i)}, n)|)$ after the precomputations explained in Lem. 10.

### 4.1.2 Boltzmann sampler

The shuffle product of languages fits well with exponential generating functions (see Theorem 7): the EGF of the shuffle product is the product of the EGF of the components. As a consequence, the exponential Boltzmann sampler for the shuffle of languages is easy to construct from the exponential Boltzmann samplers of the component languages. Below, denote by Mono-Boltz-Expo a monolithic routine that realises an exponential Boltzmann sampler for a single automaton, see [4].

The situation is not as simple for ordinary generating functions. We use a method from [8] for obtaining an ordinary Boltzmann sampler by appropriately biasing an exponential Boltzmann sampler. This is the methodology followed in Algorithm 4 below. The weight functions $u \rightarrow e^{-u} \prod_{i=1}^{K} \tilde{L}^{(i)}(zu)$ should be computed numerically rather than symbolically.

**Theorem 12.** Shuffle-Boltz-Expo($(A^{(i)}))_{i \in [K]}$, $u$) is an exponential Boltzmann sampler of parameter $u$ for $L = L^{(1)} \cup \cdots \cup L^{(K)}$ and Shuffle-Boltz($(A^{(i)}))_{i \in [K]}$, $z$) is an ordinary Boltzmann sampler of parameter $z$ for $L$.
Algorithm 3 Exponential Boltzmann sampler Shuffle-Boltz-Expo\((\mathcal{A}^{(i)})_{i \in [K]}, u)\).

Require: \(K\) DFAs \(\mathcal{A}^{(i)}\) with languages \(\mathcal{L}^{(i)}\).
Ensure: realise an exponential Boltzmann sampler for \(\mathcal{L}^{(1)} \cup \cdots \cup \mathcal{L}^{(K)}\) of parameter \(u\).
1: for \(i = 1\) to \(K\) do
2: \(w^{(i)} \leftarrow\) Mono-Boltz-Expo\((\mathcal{A}^{(i)}, u)\)
3: return Shuffle-Words\((w^{(i)}_{i \in [K]}))\).

Algorithm 4 Ordinary Boltzmann sampler Shuffle-Boltz\((\mathcal{A}^{(i)})_{i \in [K]}, z)\).
Require: \(K\) DFAs \(\mathcal{A}^{(i)}\) with languages \(\mathcal{L}^{(i)}\).
Ensure: realise an ordinary Boltzmann sampler for \(\mathcal{L}^{(1)} \cup \cdots \cup \mathcal{L}^{(K)}\) of parameter \(z\).
1: Choose \(u\) according to the weight function: \(u \mapsto e^{-u} \prod_{i=1}^{K} \hat{\mathcal{L}}^{(i)}(zu)\);
2: return Shuffle-Boltz-Expo\((\mathcal{A}^{(i)})_{i \in [K]}, zu)\).

4.1.3 Parry sampler

The following theorem ensures that a Parry sampler for the shuffle language is very easy to construct given Parry samplers for the component automata.

The Parry sampler associated with a DFA is defined via the spectral attributes of its adjacency matrix (Section 2.3). For the product automaton, spectral attributes admit compact representations, thus avoiding the explicit construction of the exponentially big adjacency matrix.

| Lemma 13. Let \(\mathcal{A} = \mathcal{A}^{(1)} \times \cdots \times \mathcal{A}^{(K)}\) be the product of \(K\) strongly connected DFAs without synchronisation. Let \(\rho, v, (\rho^{(i)})_{i \in [K]}, (v^{(i)})_{i \in [K]}\) be the spectral attributes defined as in Theorem 1. Then \(\rho = \sum_{i=1}^{n} \rho^{(i)}\) and \(v_s = \prod_{i=1}^{K} v^{(i)}_{s(i)}\) for every \(s \in Q\).

This lemma enables us to design a Parry sampler compositionally.

| Theorem 14. Given \(K\) strongly connected automata \((\mathcal{A}^{(i)})_{i \in [K]}\) without synchronised action, Algorithm 5 is a Parry sampler for the shuffle of their languages.

4.2 General case with synchronisation

In the general case with synchronisation, we rely on the decomposition of Theorem 7, as stated in the beginning of Section 4.

4.2.1 Fixed length uniform sampler

The idea of our recursive method, described in Algorithm 7 is as follows: either choose to generate a word without synchronisation that leads to a final state, or choose to generate a word without synchronisation that leads to a synchronised transition, take this transition and repeat recursively from the current state. The weight we attribute to each choice is proportional to the cardinality of the language corresponding to this choice. These cardinalities are computed in Algorithm 6.

Denote by \(\text{CompInvMat}(m)\) the complexity of inverting a square matrix of size \(m \times m\).

| Lemma 15. Algorithm 6 runs in time \(O(n \log n(\text{CompInvMat}(|Q_{\text{red}}|) + K|\Delta_{\text{red}}|))\).

| Theorem 16. Algorithm 7 is a uniform sampler that runs in linear time after precomputations made in Algorithm 6.
Then realise a Parry sampler for Proposition 18.

In this section, we consider a network of automata with synchronisations such that the product automaton is strongly connected (the case without synchronisations was treated in Section 4.1.3). As before, we work with the matrix $M(z)$ associated with the reduced automaton to avoid constructing the adjacency matrix $A$ of the product automaton. Proposition 18 is a way of stating the Perron-Frobenius theorem with $M(z)$ rather than $A$, enabling us to describe the Parry measure wrt. the reduced automaton in Theorem 19.

Proposition 18. Let $\tau$ and $\nu$ be the convergence radii and Perron vector associated to the product automaton. Then $\nu$ and the restriction $\nu_{\text{red}}$ of $\nu$ to $Q_{\text{red}}$ can be characterised wrt. $M(z)$ as follows: $\tau = \min\{z > 0 \mid \det(I - zM(z)) = 0\}; \nu_{\text{red}}$ is the unique vector such that $\nu_{\text{red}} \geq 0$ and $\tau M(\tau)\nu_{\text{red}} = \nu_{\text{red}}$.

4.2.2 Boltzmann sampler

Boltzmann sampling is obtained from the system of equations

$$L_s(z) = \tilde{L}_s(z) + z \sum_{\delta = (s, \alpha, t) \in \Delta_{\text{red}}} \tilde{L}_{\delta}(z) L_{\epsilon}(z)$$

Boltzmann sampling also applies the generic method of random sampling depicted at the beginning of this section. The procedure is described in Algorithm 8. If no synchronisation occurs (probability $\tilde{L}_s(z)/L_s(z)$), we sample a word in the shuffle of the $\tilde{A}^{(i)}_{\tilde{s},\tilde{\delta},\tilde{t}}$, using Boltzmann sampling with parameter $z$. Otherwise we uniformly choose a transition $\delta = (s, \alpha, t) \in \Delta_{\text{red}}$, generate $u$ in the shuffle of the $A^{(i)}_{s,\alpha,t}$, using Boltzmann sampling with parameter $z$, write $wu$ and repeat from $t$ to generate the rest of the word.

Theorem 17. Algorithm 8 is an ordinary Boltzmann sampler.

4.2.3 Parry sampler

In this section, we consider a network of automata with synchronisations such that the product automaton is strongly connected (the case without synchronisations was treated in Section 4.1.3).

As before, we work with the matrix $M(z)$ associated with the reduced automaton to avoid constructing the adjacency matrix $A$ of the product automaton. Proposition 18 is a way of stating the Perron-Frobenius theorem with $M(z)$ rather than $A$, enabling us to describe the Parry measure wrt. the reduced automaton in Theorem 19.

Proposition 18. Let $\tau$ and $\nu$ be the convergence radii and Perron vector associated to the product automaton. Then $\tau$ and the restriction $\nu_{\text{red}}$ of $\nu$ to $Q_{\text{red}}$ can be characterised wrt. $M(z)$ as follows: $\tau = \min\{z > 0 \mid \det(I - zM(z)) = 0\}; \nu_{\text{red}}$ is the unique vector such that $\nu_{\text{red}} \geq 0$ and $\tau M(\tau)\nu_{\text{red}} = \nu_{\text{red}}$. 

Algorithm 5 Parry sampler Shuffle-Parry($((A^{(i)})_{i \in [K]})$)

Require: $K$ strongly connected DFAs $A^{(i)}$ with languages $L^{(i)}$, spectral radii $\rho^{(i)}$, and, for each $A^{(i)}$, a Parry sampler $M^{(i)}$.

Ensure: realise a Parry sampler for $L^{(1)} \sqcup \cdots \sqcup L^{(K)}$.

1: runs $M^{(i)}$ for $i \in [K]$ in parallel to get $K$ infinite random words $(w^{(i)})_{i \in [K]}$;
2: while true do
3: choose $i$ with probability $\rho^{(i)}/(\rho^{(1)} + \cdots + \rho^{(K)})$;
4: remove from $w^{(i)}$ its first letter and write it on the output tape;
5: end while

Algorithm 6 Compo-Card($((A^{(i)})_{i \in [K]}, n)$) (precomputation of cardinalities for Compo-Unif).

Require: $K$ DFAs $A^{(i)}$ and a natural integer $n$.

Ensure: compute and store every cardinalities used in Compo-Unif($((A^{(i)})_{i \in [K]}, s, n)$).

1: for $s \in Q_{\text{red}}$ do
2: Shuffle-Card($((A^{(i)}_{s,i})),_{i \in [K]}, n)$; (this implicitly defines $\tilde{L}(z) \mod z^{n+1}$)
3: for $\delta \in \Delta_{\text{red}}$ do
4: Shuffle-Card($((A^{(i)}_{s,i})),_{i \in [K]}, n)$; (this implicitly defines $M(z) \mod z^{n+1}$)
5: Compute $L(z) \mod z^{n+1}$ by solving $L(z) = \tilde{L}(z) + zM(z)L(z)$ with all generating functions and operations modulo $z^{n+1}$.
Algorithm 7 Uniform sampler Compo-Unif((A^{(i)})_{i \in [K]}, s, n).

Require: K DFAs A^{(i)}, a natural integer n and a state s \in Q_{\text{red}} and cardinalities computed by Compo-Card((A^{(i)})_{i \in [K]}, n).

Ensure: return a word in L_s of length n uniformly at random.
1: with probability \frac{l_{s,n}}{l_{s,n}} return Shuffle-Unif((A^{(i)})_{i \in [K]}, n);
2: choose m with weight \sum_{\delta=(s,\alpha,t) \in \Delta_{\text{red}}} l_{\delta,m-1} l_{t,n-m} / \sum_{m=1}^{\infty} \sum_{\delta=(s,\alpha,t) \in \Delta_{\text{red}}} l_{\delta,m-1} l_{t,n-m};
3: choose \delta = (s, \alpha, t) \in \Delta_{\text{red}} with probability \frac{l_{\delta,m-1} l_{t,n-m}}{\sum_{m=1}^{\infty} \sum_{\delta=(s,\alpha,t) \in \Delta_{\text{red}}} l_{\delta,m-1} l_{t,n-m}};
4: w \leftarrow Shuffle-Unif((A^{(i)})_{i \in [K]}, m);
5: return w :: \alpha :: Compo-Unif((A^{(i)})_{i \in [K]}, t, n - m).

Algorithm 8 Ordinary Boltzmann sampler Compo-Boltz((A^{(i)})_{i \in [K]}, s, z).

Require: K DFAs A^{(i)}, a state s \in Q_{\text{red}} and a parameter z.

Ensure: realises a ordinary Boltzmann sampler of parameter z for the language L_s.
1: With probability \frac{L_s(z)}{L_s(z)} return Shuffle-Boltz((A^{(i)})_{i \in [K]}, z);
2: Choose \delta = (s, \alpha, t) \in \Delta_{\text{red}} with probability \frac{\tilde{L}_\delta(z) L_t(z)}{\sum_{\delta=(s,\alpha,t) \in \Delta_{\text{red}}} \tilde{L}_\delta(z) L_t(z)});
3: return Shuffle-Boltz((A^{(i)})_{i \in [K]}, z) :: \alpha :: Compo-Boltz((A^{(i)})_{i \in [K]}, t, z).

◮ Theorem 19. Algorithm 9 is a Parry sampler of infinite words starting from the input state s \in Q_{\text{red}}.

5 Conclusion and further work

In this paper, we propose several algorithms for uniformly sampling words recognised by networks of automata. For the purely interleaved case, with no synchronisation, the sampling algorithms are polynomial in the size of the component automata, while previous known algorithms were exponential (since they were polynomial on the exponentially big product automaton). For the general case where synchronisations occur on shared actions, our methods are efficient with respect to the reduced automaton (the product automaton where interleaved actions are removed).

We plan to implement the different algorithms presented here and apply them on benchmarks. For instance, we could use the benchmarks of [9] and [19]. We would also like to incorporate our uniform sampling into a Monte-Carlo model checker.

The recursive methods for words of a fixed length n that we presented here have a bit complexity which is essentially n times bigger than their arithmetic complexity since the cardinalities we compute grow exponentially with n and each one needs a linear amount of bits to be stored. We think that this extra factor n can be avoided using a divide-and-conquer paradigm akin to that of [5]. Further work to deal with interleaving and synchronisations has to be done though.

In the context of Monte-Carlo model checking of Büchi properties [13, 20] the meaningful objects to sample are accepting lassos (an accepting lasso is a path followed by an elementary cycle that visits accepting states). We would like to design uniform sampling of accepting lassos for networks of automata by building on top of the present work and on [20].
Algorithm 9 Parry sampler $\text{Parry}((\mathcal{A}^{(i)})_{i \in [K]}, s)$

Require: $K$ DFAs $\mathcal{A}^{(i)}$ and a state $s \in \mathcal{Q}_{\text{red}}$.
Ensure: realises a Parry sampler of infinite words from $s$.

1: choose $\delta = (s, \alpha, t) \in \Delta_{\text{red}}$ with probability $\frac{\tilde{L}_\delta(t)}{v_t / v_s}$;
2: return $\text{Shuffle-Boltz}(\tilde{L}_\delta, \alpha) :: \alpha :: \text{Parry}((\mathcal{A}^{(i)})_{i \in [K]}, t)$.

References


Tractability of Separation Logic with Inductive Definitions: Beyond Lists

Taolue Chen¹, Fu Song², and Zhilin Wu³

1 Department of Computer Science and Information Systems, Birkbeck, University of London, UK; and
State Key Laboratory of Novel Software Technology, Nanjing University, China
2 School of Information Science and Technology, ShanghaiTech University, China
3 State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, China

Abstract

In 2011, Cook et al. showed that the satisfiability and entailment can be checked in polynomial time for a fragment of separation logic that allows for reasoning about programs with pointers and linked lists. In this paper, we investigate whether the tractability results can be extended to more expressive fragments of separation logic that allow defining data structures beyond linked lists. To this end, we introduce separation logic with a simply-nonlinear compositional inductive predicate where source, destination, and static parameters are identified explicitly (SLID). We show that if the inductive predicate has more than one source (destination) parameter, the satisfiability problem for SLID becomes intractable in general. This is exemplified by an inductive predicate for doubly linked list segments. By contrast, if the inductive predicate has only one source (destination) parameter, the satisfiability and entailment problems for SLID are tractable. In particular, the tractability results hold for inductive predicates that define list segments with tail pointers and trees with one hole.

1998 ACM Subject Classification F.4.1 Mathematical Logic

Keywords and phrases Separation logic, Inductive definitions, Satisfiability, Entailment

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2017.37

1 Introduction

Separation Logic (SL, [23, 25]), an extension of Hoare logic, is a well-established formalism for the verification of heap manipulating programs. SL features a separating conjunction operator and inductive predicates, which allow to express how data structures are laid out in memory in a succinct way. Since its introduction, various verification tools based on separation logic have been developed. A notable one among them is the INFER tool [9], which was acquired by Facebook in 2013 and has been actively used in its development process [10]. To enable program verification with SL assertions in a Hoare-logic style, it is vital to be able to check satisfiability and entailment of SL formulae. Unfortunately, entailment checking in full SL is undecidable [11]. Thus one has to either resort to heuristics or consider restricted, decidable fragments. We focus on the latter approach in this paper. Earlier efforts, including the theoretical work [2] which leads to the tool SMALLFoot [3], considered SL with primitives of pointers and the hardwired singly linked list segments. A remarkable result on

* This work was partially supported by the UK EPSRC grant (EP/P00430X/1), the European CHIST-ERA project SUCCESS, the NSFC grants (61472474, 61402179, 61532019, 61662035, 61572478).
Tractability of Separation Logic with Inductive Definitions: Beyond Lists

this fragment was given by Cook et al. [14], who developed a polynomial-time algorithm for the satisfiability and entailment problems based on graph reasoning. This has also led to the tool SeLoger [19].

Hardwired inductive predicates have obvious limitations. More recent research has been focusing on automated reasoning about generic user-defined inductive predicates inside the framework of SL. This class of logic is usually referred to as Separation Logic with Inductive Definitions (SLID). Because of their generality, satisfiability and entailment checking are usually much more difficult. Nevertheless, notable progress has been made in this direction. For instance, Iosif et al. showed decidability of satisfiability and entailment for an expressive fragment of SLID by a reduction to Monadic Second Order Logic on graphs with bounded tree width [20]. Later on, they reduced the entailment problem to the language inclusion problem of tree automata, and implemented the tool SLIDE [21]. Brotherston et al. showed that the satisfiability problem (but not entailment) for a very expressive fragment of SLID is ExpTime-complete [6]. In the same paper, they also proved that the satisfiability problem becomes NP-complete if the arities of all predicates are bounded by a constant.

Contributions. We are mostly interested in tractable (polynomial-time) satisfiability and entailment checking for SLID. Our strategy is to generalise the graph reasoning initiated in [14], which currently is limited to singly linked lists only. To this end, we concentrate on a class of simply-nonlinear compositional inductive predicates, where the source, destination, and static parameters are identified explicitly. This class of inductive predicates enjoys some nice compositional properties introduced in [17], which makes, for example, the entailment problem easier to solve. The main question is whether the satisfiability and entailment checking for the resulting SLID fragment, denoted by SLID_{SNC}, can be done in polynomial time. We obtain both positive and negative results in this regard.

1. We show that, with an inductive predicate $dllseg$ for doubly linked list segments, the satisfiability problem for SLID_{SNC}$[dllseg]$ is NP-complete. From the perspective of data structures, doubly linked list segments are perhaps the minimal extension of singly linked list segments, and thus one would have expected the tractability result for doubly linked list segments by a straightforward adaptation of the method for singly linked list segments by Cook et al. in [14]. Our result is among the strongest intractability results for the satisfiability problem for SLID so far. Note that the NP-hardness reduction in [6] requires four predicates, while our reduction uses only one predicate $dllseg$. However, it should be noted that there are at most three parameters for predicates in [6], but $dllseg$ has four parameters (i.e., two source and destination parameters respectively). The result suggests that SLID_{SNC} becomes intractable in general if inductive predicates with more than one source (destination) parameter are allowed.

2. We show that both the satisfiability and the entailment checking can be done in polynomial time if the inductive predicate has only one source (destination) parameter. Remarkably, the fragment covers linear data structures (e.g., list segments with tail pointers), as well as non-linear data structures (e.g., trees with one hole). To our best knowledge, this is one of the first tractability results of SLID for a class of user-defined inductive predicates. (Independently, [8] studied a different class mostly on the model checking problem; cf. related work for comparison.)

To achieve this, we generalise the graph-theoretic techniques developed in [14], in particular, the concept of graph homomorphisms, to non-linear structures. Compared to [14], the graph representations of the SLID_{SNC} formulae have a considerably more involved structure, e.g., the unique simple path property between a pair of nodes in [14] is lost here.
An additional intricacy is that we consider the classical semantics of SLID$_{SNC}$ formulae rather than the (less general) intuitionistic semantics in [14]. The authors of [14] did briefly mention that, to tackle the classical semantics, extra conditions—although not specified—should be added to ensure that all arcs are covered in the graph homomorphism. They, however, suggested that “the details are messy and deferred to a full version.” We concur with these insights. As singly linked list segments are a (very) special case of our inductive predicates, the decision procedure in this paper provides a complete account for the entailment checking under the classical semantics.

**Related work.** There is a vast literature on separation logic, and we will have to focus on work which is the most relevant to us. In particular, we only cover the work on SLID, and skip many results on, for instance, first-order separation logic (without inductive definitions) [4, 15]. Antonopoulos et al. established some fundamental decidability and complexity results for the entailment problem of SLID [1]. In particular, they showed that deciding the entailment $\varphi \models \psi$ for SLID is intractable where $\psi$ may contain existentially quantified variables, even for a single predicate of list segments. This result is largely incomparable to us, since existential quantified variables are disallowed in our setting. The work [5, 13] took the cyclic-proof approach which is based on induction on the paths of proof trees, but the decision procedures therein are incomplete in general. Brotherston et al. investigated array separation logic and obtained some complexity results [7]: They showed that the satisfiability is NP-complete, entailment is decidable with high complexity, and bi-abduction is in NP. Recently, [8] gave another tractable fragment of SLID, but mostly focused on the model checking problem. The tractable fragment of [8], defined by three constraints MEM (“Memory-consuming”), CV (“Constructively valued”) and DET (“Deterministic”), is incomparable to ours: “trees with one hole” is in our tractable fragment, while it does not belong to MEM+CV+DET, as the DET constraint fails; however, an MEM+CV+DET formula may contain several different inductive predicates, while an $\text{SLID}_{SNC}$ formula allows at most one inductive predicate. We also mention recent work considering decision procedures for SLID extended with data and size constraints [12, 18, 26, 22, 24, 27].

In the sequel, we discuss [18] and [16] which are technically more related to this paper. Strictly speaking, [18] considered SLID extended with data constraints, so here we limit the comparison to the data-free part. First of all, [18] tackled linear structures, while here we focus on more general non-linear structures (e.g., trees). More importantly, the decision procedures in [18] are intractable, even when specialised to the data-free setting. Indeed, for both satisfiability and entailment, only NP upper-bounds can be obtained. The current work improves [18] by giving polynomial-time algorithms for even non-linear structures. In particular, for satisfiability, we extend the graph-theoretic approach in [14]: for entailment, we give a new definition of graph homomorphism, which is much more involved than that in [18], but yields an efficient checking algorithm.

[16] considered a fragment of SLID where nested lists and skip lists are expressible, and provided an incomplete decision procedure for the entailment problem by generalising the graph homomorphism in [14]. A more technical comparison is deferred to Section 4.2.

2 Preliminaries

For $n \in \mathbb{N}$, $[n] := \{1, \ldots, n\}$. We assume a set of variables $\text{Vars}$ ranged over by $E, F, X, Y, \ldots$, and a set of locations $\text{L}$ typically ranged over by $l, l', \ldots$. We assume a designated variable $\text{nil} \in \text{Vars}$ (for the “null” value of pointers in programs) and $\text{L}$ contains a special element $\text{null}$.
Moreover, we assume a set of fields $F$ ranged over by $f, f', \ldots$. We focus on separation logic with a \textit{simply-nonlinear compositional} inductive predicate (denoted by $\SLID_{\text{SNC}}[P]$):

$$
\Pi := E = F \mid E \neq F \mid \Pi \land \Pi, \\
\Sigma := \text{emp} \mid E \mapsto \rho \mid P(E, E'; F, F'; B) \mid \Sigma \ast \Sigma,
$$

where $\rho$ is a tuple of elements from $F \times \text{Vars}$, and $P(E, E'; F, F'; B)$ is a simply-nonlinear compositional inductive predicate whose definition will be specified shortly.

The formulae $\Pi$ and $\Sigma$ are called the \textit{pure} and \textit{spatial} formulae respectively. Atomic formulae of the form $E \mapsto \rho$ and $P(E, E'; F, F'; B)$ are called the \textit{points-to} and \textit{predicate} atoms respectively. For an $\SLID_{\text{SNC}}[P]$ formula $\phi$, we use $\text{Vars}(\phi)$ to denote the set of variables occurring in $\phi$, in addition, we use $\text{Atoms}(\phi)$ to denote the set of points-to or predicate atoms occurring in $\phi$. We require that the predicate $P$ in $\SLID_{\text{SNC}}[P]$ is \textit{simply-nonlinear compositional}, that is, of the following form:

1. The parameters of $P$ are divided into three categories: the \textit{source} parameters $E, E'$, the \textit{destination} parameters $F, F'$, and the \textit{static} parameters $B$. (Note that $E', F', B$ denote a vector of variables.) We require that the source and the destination parameters are \textit{symmetric}, in particular, they must be of the same length. A predicate $P$ is usually denoted by $P(E, E'; F, F'; B)$ to highlight the parameters.

2. $P(E, E'; F, F'; B)$ is defined by a set of rules including:

   - base rule: $P(E, E'; F, F'; B) := E = F \land E' = F' \land \text{emp}$,
   - inductive rule:
     $$P(E, E'; F, F'; B) := \exists X. E \mapsto [(f_1, Y_1), \ldots, (f_k, Y_k)]* \times P(X_0, X_0'; F, F'; B) \ast P(X_1, X_1'; \text{nil}, B) \ast \cdots \ast P(X_l, X_l'; \text{nil}, B),$$

such that

- $X_0, \ldots, X_l$ are mutually distinct variables and $X = \{X_0, \ldots, X_l\}$,
- $Y_1, \ldots, Y_k \in E' \cup B \cup X \cup \{\text{nil}\}$ and each variable from $E' \cup B \cup X$ occurs exactly once in $E \mapsto [(f_1, Y_1), \ldots, (f_k, Y_k)]$, 
- for each $i : 0 \leq i \leq l$, $X_i \subseteq \{E\} \cup X$.

Note that above, in $Y_1, \ldots, Y_k \in E' \cup B \cup X \cup \{\text{nil}\}$ or $X_i \subseteq \{E\} \cup X$, $E'$ should be interpreted as the \textit{set} of variables occurring in $E'$, similarly for $B, X, X_i$. We assume that, if there are multiple inductive rules, the same set of fields $\text{Fids}(P)$ is used. In addition, we define the set $\text{PFids}(P)$ of \textit{principal fields}, as the set of fields $f \in \text{Fids}(P)$ such that there is an inductive rule of $P$ where $(f, Z)$ occurs for some $Z \in X$, and the set $\text{Afields}(P)$ of \textit{auxiliary fields}, as the set of fields $f \in \text{Fids}(P)$ such that there is an inductive rule of $P$ where $(f, Z')$ occurs for some $Z' \in E' \cup B \cup \{\text{nil}\}$. We assume that $\text{PFids}(P) \cap \text{Afields}(P) = \emptyset$, which implies that $\text{Fids}(P) = \text{PFids}(P) \cup \text{Afields}(P)$.

In the base or inductive rule, the formula on the left (resp. right) of $\vdash$ is called the \textit{head} (resp. \textit{body}) of the rule. For each predicate $P(E, E'; F, F'; B)$, there is exactly one base rule, and in each inductive rule, each destination parameter from $F, F'$ occurs exactly once in the body of the rule, namely, in $P(X_0, X_0'; F, F'; B)$. We use $\mathcal{P}$ to denote the set of simply-nonlinear compositional inductive predicates. Moreover, we use $\mathcal{P}_1$ to denote the set of inductive predicates $P(E; F; B) \in \mathcal{P}$, i.e., the set of inductive predicates with a single source resp. destination parameter.

\textbf{Semantics.} Each $\SLID_{\text{SNC}}[P]$ formula is interpreted on \textit{states}. Formally, given a finite set of fields $F \subseteq F$, a \textit{state} is a pair $(s, h)$, where
\[ s \text{ is a stack which is a partial function } \text{Vars} \rightarrow \text{L} \text{ such that } \text{dom}(s) \text{ is finite, } \text{nil} \in \text{dom}(s), \text{ and } s(\text{nil}) = \text{null}, \]

\[ h \text{ is a heap which is a partial function } \text{L} \times \text{F} \rightarrow \text{L} \text{ such that } \text{dom}(h) \text{ is finite, } h(\text{null}, f) \text{ is undefined for each } f \in \text{F}, \text{ and for each } l \in \text{L}, \text{ if } h(l, f) \text{ is defined for some } f \in \text{F}, \text{ then } h(l, f') \text{ is defined for each } f' \in \text{F}. \]

We use States to denote the set of states. For a heap h, we use \( \text{ldom}(h) \) to denote the set of locations \( l \in \text{L} \) such that \( h(l, f) \) is defined for some \( f \in \text{F} \). Note that \( \text{null} \notin \text{ldom}(h) \). We write \( h_1 \neq h_2 \) if \( \text{ldom}(h_1) \cap \text{ldom}(h_2) = \emptyset \), in which case we write \( h_1 \uplus h_2 \) for the union of \( h_1 \) and \( h_2 \). A heap \( h_1 \) is called a subheap of \( h_2 \) if \( \text{ldom}(h_1) \subseteq \text{ldom}(h_2) \) and for each \( l \in \text{ldom}(h_1) \) and \( f \in \text{F}, h_1(l, f) = h_2(l, f) \).

Let \((s, h) \in \text{States} \) and \( \varphi \) be an \( \text{SLDSNC}[P] \) formula. Then the semantics of \( \varphi \) is defined as follows,

\[ (s, h) \models E = F \text{ (resp. } (s, h) \models E \neq F \text{) if } s(E) = s(F) \text{ (resp. } s(E) \neq s(F)), \]

\[ (s, h) \models \Pi_1 \land \Pi_2 \text{ if } (s, h) \models \Pi_1 \text{ and } (s, h) \models \Pi_2, \]

\[ (s, h) \models \text{emp} \text{ if } \text{ldom}(h) = \emptyset; \]

\[ (s, h) \models E \Rightarrow \{(f_1, X_1), \ldots , (f_k, X_k)\} \text{ if } \text{ldom}(h) = s(E), \text{ and for each } i : 1 \leq i \leq k, h(s(E), f_i) = s(X_i); \]

\[ (s, h) \models \text{P}(E, E', F, F', B) \text{ if } (s, h) \in \text{P}(E, E', F, F', B), \]

\[ (s, h) \models \Sigma_1 \land \Sigma_2 \text{ if } h = h_1 \uplus h_2, (s, h_1) \models \Sigma_1 \text{ and } (s, h_2) \models \Sigma_2, \]

where the semantics of predicate \( \text{P}(E, E', F, F', B) \) is given by the least fixpoint of a monotone operator constructed from the body of rules for \( P \). More concretely, assume that \( P \) contains \( m \) inductive rules, each of which is denoted by \( R_i \) and is of the form \( R_i : \text{P}(E, E', F, F', B) := \Theta_i \) (i.e., \( \Theta_i \) is the body of \( R_i \)). For each \( R_i \) (where \( i \in [m] \)), we define a monotone operator \( \tau : 2^{\text{States}} \rightarrow 2^{\text{States}} \) as follows: \( \tau_i(S) := \{(s, h) \mid (s, h) \models_{P, S} \Theta_i \} \), where \( \models_{P, S} \) is the satisfaction relation defined above, except that \( P \) is interpreted by \( S \), that is, \((s, h) \models \text{P}(E, E', F, F', B) \text{ if } (s, h) \in S \). We finally define \( \text{S}(P) := \mu S(\tau(S)) \).

**Example 1.** We use predicates lseg, dllseg, tseg, and th to exemplify our definitions, which represent list segments, doubly linked list segments, list segments with tail pointers, and binary trees with one hole, respectively.

\[
\text{lseg}(E; F) \quad ::= \quad E = F \land \text{emp},
\]

\[
\text{dllseg}(E; P; F; L) \quad ::= \quad \exists X. \ E \Rightarrow (\text{next}, X) \land \text{lseg}(X, F).
\]

\[
\text{dllseg}(E; P; F; L) \quad ::= \quad \exists X. \ E \Rightarrow (\text{next}, X) \land \text{dllseg}(X, E; F; L).
\]

\[
\text{tseg}(E; F; B) \quad ::= \quad E = F \land \text{emp},
\]

\[
\text{tseg}(E; F; B) \quad ::= \quad \exists X. \ E \Rightarrow (\text{next}, X) \land \text{dllseg}(X; F; B).
\]

\[
\text{th}(E; F) \quad ::= \quad E = F \land \text{emp},
\]

\[
\text{th}(E; F) \quad ::= \quad \exists X, Y. \ E \Rightarrow [(\text{left}, X), (\text{right}, Y)] \land \text{th}(X; \text{nil}) \land \text{th}(Y; F),
\]

\[
\text{th}(E; F) \quad ::= \quad \exists X, Y. \ E \Rightarrow [(\text{left}, X), (\text{right}, Y)] \land \text{th}(E; F) \land \text{th}(Y; \text{nil}).
\]

For \( P \in \mathcal{P} \), we investigate the following two decision problems:

**Satisfiability.** Given an \( \text{SLDSNC}[P] \) formula \( \varphi \), decide whether there is a state \((s, h) \models \varphi.\)

**Entailment.** Given two \( \text{SLDSNC}[P] \) formulae \( \varphi \) and \( \psi \) with \( \text{Vars}(\psi) \subseteq \text{Vars}(\varphi) \), decide whether \( \varphi \models \psi \), i.e., for every state \((s, h) \models \varphi \) implies \((s, h) \models \psi \).

The inductive predicates defined above fall into the category of *compositional inductive predicates* [17] and admit the composition lemma, i.e., \( P(E_1, E_1'; E_2, E_2'; B) \land P(E_2, E_2'; E_3, E_3'; B) \models P(E_1, E_1'; E_2, E_2'; E_3, E_3'; B) \) holds, which is crucial for the decision procedure of entailment.
In literature, there is usually a distinction of the classical semantics and the intuitionistic semantics. The aforementioned semantics for SLID$_{SNC}[P]$ formulae are called the classical semantics. In the intuitionistic semantics, pure and spatial formulae are interpreted in the same way as the classical one. The only difference is, in the intuitionistic semantics, we have, for an SLID$_{SNC}[P]$ formula $\phi = \Sigma \wedge \Pi$, $(s, h) \models \phi$ if there is a subheap $h_1$ of $h$, $(s, h_1) \models \Pi$ and $(s, h_1) \models \Sigma$. Henceforth, when necessary, we write $\models$ to denote the satisfiability relation under the classical semantics, and accordingly, $\models_i$ to denote the satisfiability relation under the intuitionistic semantics. The two semantics are equivalent for the satisfiability problem, but differ in the entailment problem. For instance, $lseg(E_1; E_2) \ast lseg(E_2; E_3) \models \ast lseg(E_1; E_2)$ does not hold, while $lseg(E_1; E_2) \ast lseg(E_2; E_3) \models_i lseg(E_1; E_2)$ does hold.

**Graph-theoretic notions.** We shall work extensively upon an arc-labeled directed graph equipped with a symmetry relation. Here we collect some notations on this matter. For simplicity, we ignore the arc-labels and symmetric relations here, since they are irrelevant right now. In general, a directed graph in $G$ is a pair $(N, E)$, where $N$ is a finite set of nodes, $E \subseteq N \times N$ is a set of directed arcs. For an arc $e = (v, w) \in E$, $v$ and $w$ are called the source and destination node of $e$ and sometimes $w$ is called a successor of $v$ and $v$ is called a predecessor of $w$.

A path in $G$ is a sequence of consecutive directed arcs in $G$. A node $w$ is reachable from $v$ if there is a path from $v$ to $w$ (note that $v$ is reachable from $v$ itself). In this case, we also say that $v$ is an ancestor of $w$. A cycle is a path where the starting node and the ending node are equal. A path is simple if no nodes occur twice on the path. A cycle is simple if all nodes are distinct, except the ending node. For a node $v$ and a directed arc $e = (v', w')$, $e$ is said to be reachable from $v$ if $v'$ is reachable from $v$, and $e$ is said to be co-reachable from $v$ if $v$ is reachable from $w'$. We use $R_G(v)$ to denote the set of arcs that are reachable from $v$.

For a graph $G$, let $N(G)$ denote the set of nodes in $G$. For a set of arcs $E' \subseteq E$, let $N(E')$ denote the set of all source or destination nodes of arcs in $E'$. For $N' \subseteq N$, the subgraph of $G$ induced by $N'$, denoted by $G[N']$, is $(N', E \cap (N' \times N'))$. On the other hand, for $E' \subseteq E$, the subgraph of $G$ induced by $E'$, denoted by $G[E']$, is $(N', E')$.

A connected component (CC) $C$ (resp. strongly connected component, SCC, $S$) of $G$ is said to be nontrivial if $C$ (resp. $S$) contains at least one arc.

### 3 Intractability: Doubly linked list segments

Let $\phi = \Pi \wedge \Sigma$ be an SLID$_{SNC}[dllseg]$ formula, where $dllseg$ is given in Example 1. We shall show the intractability of deciding the satisfiability of SLID$_{SNC}[dllseg]$ formulae. The intractability is proved by a reduction from 3SAT. It turns out that the reduction works even for a restricted class of SLID$_{SNC}[dllseg]$ formulae $\Pi \wedge \Sigma$ where $\Pi$ only contains inequalities and $\Sigma$ only contains predicate atoms. To ease the presentation of the reduction, we will introduce a graphical representation for this restricted class of SLID$_{SNC}[dllseg]$ formulae. For an SLID$_{SNC}[dllseg]$ formula $\phi$, recall that $\text{Vars}(\phi)$ and $\text{Atoms}(\phi)$ denote the set of variables and atoms in $\phi$. We construct a graph $G_\phi = (N, A, \mathcal{E}, \mathcal{L}, R_\phi)$ as follows.

- $N = \text{Vars}(\phi)$ and $A = \text{Atoms}(\phi)$.
- $\mathcal{E} \subseteq N \times A \times N$ is the set of arcs and $\mathcal{L}$ is the arc-labeling function, defined as: for each predicate atom $a = dllseg(E, P; F, L)$ in $\text{Atoms}(\phi)$, there are two arcs $e = (E, a, F) \in \mathcal{E}$ and $e' = (L, a, P)$ with $\mathcal{L}(e) = (a, +)$ and $\mathcal{L}(e') = (a, -)$ respectively.
- $R_\phi = \{(E, F), (F, E) \mid \Pi \models E \neq F\}.$
For convenience, we usually write \( E \xrightarrow{\ell} F \) for an arc \((E, F)\) labeled by \( \ell \). Clearly, \( E \) satisfies that, for each predicate atom \( a \), there are exactly two arcs in \( \mathcal{G} \), labeled by \((a, +)\) and \((a, -)\) respectively. We usually say that these two arcs are dual and have the opposite poles. All graphs for restricted \( \text{SLID}_{\text{SNC}}[\text{dllseg}] \) formulae defined as above form a class of (restricted) \( \text{SLID}_{\text{SNC}}[\text{dllseg}] \) graphs.

From each restricted \( \text{SLID}_{\text{SNC}}[\text{dllseg}] \) graph \( \mathcal{G}_\phi = (\mathcal{N}, \mathcal{A}, \mathcal{E}, \mathcal{L}, R_\phi) \), we can recover a (restricted) \( \text{SLID}_{\text{SNC}}[\text{dllseg}] \) formula \( \varphi_G = \Pi_G \land \Sigma_G \), where \( \Pi_G = \bigwedge_{(E, F) \in R_\phi} E \neq F \) and \( \Sigma_G = \bigwedge_{a \in A} a \in L \xrightarrow{(a, +)} (a, -) \) graphs can be easily transformed to each other, we will use them interchangeably.

We are ready to present the reduction from the 3SAT Problem. Let \( \Psi \) be a CNF formula with clauses \( C_1, \ldots, C_m \) over a set of variables \( \{x_1, \ldots, x_n\} \). For each \( j \in [m] \), \( C_j \) is a disjunction of at most three literals, that is, \( C_j = l_{j,1} \lor l_{j,2} \lor l_{j,3} \), where for each \( k = 1, 2, 3 \), \( l_{j,k} = x_i \) or \( \neg x_i \) for some \( i \in [n] \). Without loss of generality, we assume that for each \( i \in [n] \) and \( j \in [m] \), \( x_i \) occurs at most once in \( C_j \).

We introduce a set of atoms \( \mathcal{A}_\Psi = \{x_{i,j}, \overline{x_{i,j}} \mid x_i \text{ occurs in } C_j\} \). Furthermore, for each \( j \in [m] \), each literal \( l_{j,k} \) in \( C_j \) \( (k \in \{1, 2, 3\}) \) is associated with an atom \( x_{i,j,k} \) defined as follows: if \( l_{j,k} = x_i \), then \( l_{j,k} = x_{i,j,k} \); otherwise, \( l_{j,k} = \overline{x_{i,j,k}} \). We then construct \( \text{SLID}_{\text{SNC}}[\text{dllseg}] \)-graphs for variables and clauses respectively.

For each variable \( x_i \), assume that \( x_i \) appears in \( C_{j_1}, C_{j_2}, \ldots, C_{j_{k_i}} \) \( (1 \leq j_1 < \cdots < j_{k_i} \leq m) \), we have the following arcs: (1) \( E_{i,j_1} X_{i,1}^1 E_{i,j_1} \xrightarrow{(j_1,1, +)} X_{i,1}^1 \), \( E_{i,j_1} \xrightarrow{(j_1,1, -)} Y_{i,1}^s \) for \( s : 1 \leq s < k_i \). Moreover, for convenience, we write \( F_i \) for \( X_{i,k_i}^s \) and \( F_i' \) for \( Y_{i,k_i}^s \). Then we set \( (F_i, F_i') \), \((F_i, F_i') \in R_\phi \).

For each clause \( C_j \), if \( C_j = l_{j,1} \lor l_{j,2} \lor l_{j,3} \), then we have arcs \( G_{j} (l_{j,1}, - \to H_j) \), \((l_{j,2}, - \to H_j) \in R_\phi \). The constructions for the cases \( C_j = l_{j,1} \land C_j = l_{j,2} \lor l_{j,3} \) are similar.

Finally, \( \mathcal{G}_\Psi \) comprises the collection of \( \text{SLID}_{\text{SNC}}[\text{dllseg}] \)-graphs for all variables \( x_i \) \( (i \in [n]) \) and all clauses \( C_j \) \( (j \in [m]) \).

It is easy to see that in \( \mathcal{G}_\Psi \), for each pair of atoms \( x_{i,j} \) or \( \overline{x_{i,j}} \), one of them appears twice, but the other occurs exactly once. To make \( \mathcal{G}_\Psi \) a (restricted) \( \text{SLID}_{\text{SNC}}[\text{dllseg}] \)-graph defined above, we can simply add some additional isolated arcs. For instance, if \( (x_{i,j}, +) \) occurs, but \( \neg x_{i,j} \) does not \( (i,j) \) in \( \mathcal{G}_\Psi \) (because \( \neg x_{i,j} \) but not \( x_i \) occurs in \( C_j \)), we add an arc \( G_{j} (l_{j,1}, - \to H_j) \), \((l_{j,2}, - \to H_j) \) where \( G_{j} \), \( H_j \) are fresh variables.

**Example 2.** The following example illustrates the intuition of the reduction. Suppose \( \Psi = (x_1 \lor x_2) \land (\neg x_1 \lor \neg x_2) \) and the graph \( \mathcal{G}_\Psi \) is illustrated in Figure 1, where dashed lines represent \( R_\phi \). In a model \((s, h)\) of \( \mathcal{G}_\Psi \), since \( (G_1, H_1) \in R_\phi \), at least one of \( \mathcal{G}_1 \), \( \mathcal{G}_2 \), say, \( \mathcal{G}_1 \), must be assigned with a nonempty subheap. By the semantics of \(*\), at least one of the two paths from \( E_1 \) to \( F_1 \) and from \( E_1 \) to \( F'_1 \) must be empty. Since \( (\mathcal{G}_1, +) \) appears in the path from \( E_1 \) to \( F_1 \) and \( \mathcal{G}_1 \) is nonempty, we deduce that the path from \( E_1 \) to \( F'_1 \) has to be assigned with an empty subheap. Accordingly, since \( (\mathcal{G}_1, +) \) occurs in the path from \( E_1 \) to \( F'_1 \), the atom \( (\overline{\mathcal{G}_1}, +) \) is assigned with an empty subheap. In addition, from \( G_2 \) \( \xrightarrow{(\mathcal{G}_2, - \to H_2) \) \( \mathcal{G}_2 \) \( \xrightarrow{(\overline{\mathcal{G}_2}, - \to H_2) \) \( H_2 \).
As a convention, whose formal definition is omitted here due to the page limit. It is easy to verify that

Theorem 3. The satisfiability problem for \( \text{SLID}_{\text{SNC}}[\text{dllseg}] \) is NP-complete.

4 Tractability: Beyond list segments

In this section, we show that if \( P \in \mathcal{P}_1 \), then \( \text{SLID}_{\text{SNC}}[P] \) is tractable for both satisfiability and entailment. We fix a predicate \( P(E; F; B) \) tractable for \( \mathcal{P}_1 \).

4.1 Satisfiability

For an \( \text{SLID}_{\text{SNC}}[P] \) formula \( \varphi \), we construct a graph \( \mathcal{G}_\varphi \) to represent \( \varphi \). Specifically, \( \mathcal{G}_\varphi \) is a tuple \((\text{Vars}(\varphi), \text{Atoms}(\varphi), \mathcal{N}_\varphi, \mathcal{E}_\varphi, \mathcal{L}_\varphi, \mathcal{R}_\varphi)\) defined as follows.

\( \mathcal{N}_\varphi = \{ [E] \mid E \in \text{Vars}(\varphi) \} \) such that \([E]\) is the equivalence class of \( E \) under (the equivalence relation) \( \sim_{\Pi} \) where \( E' \sim_{\Pi} F' \) if \( \Pi \models E' = F' \).

\( \mathcal{E}_\varphi \) is the set of arcs, and \( \mathcal{L}_\varphi \) is the arc-labeling function, defined as follows:

- for each points-to atom \( a = E \mapsto [(f_1, F_1), \ldots, (f_k, F_k)] \) in \( \text{Atoms}(\varphi) \) and each \( i \in [k] \) with \( f_i \in \text{PFlds}(P) \), \( \mathcal{E}_\varphi \) contains a field-labeled arc \( e = ([E], [F]) \) with \( \mathcal{L}_\varphi(e) = (f_i, a) \);

- for each predicate atom \( a = P(E; F; B) \) in \( \text{Atoms}(\varphi) \), \( \mathcal{E}_\varphi \) contains a predicate-labeled arc \( e = ([E], [F]) \) with \( \mathcal{L}_\varphi(e) = (P, a) \);

In both cases, \( a \) is the atom associated with \( e \), denoted by \( \text{atom}(e) \). In some places later on, in order to emphasise that \( e \) is from \( \mathcal{G}_\varphi \), we also use \( \text{atom}_\varphi(e) \), instead of \( \text{atom}(e) \).

\( \mathcal{R}_\varphi = \{ ([E], [F]), ([F], [E]) \mid \Pi \models E \neq F \} \).

As a convention, \( \bot \) represents the graph for an unsatisfiable \( \text{SLID}_{\text{SNC}}[P] \) formula.

All graphs for \( \text{SLID}_{\text{SNC}}[P] \) formulae defined as above form a class of \( \text{SLID}_{\text{SNC}}[P] \) graphs, whose formal definition is omitted here due to the page limit. It is easy to verify that \( \text{SLID}_{\text{SNC}}[P] \) formulae and \( \text{SLID}_{\text{SNC}}[P] \) graphs are equivalent, namely, they can be transformed into each other. Hence we will use them interchangeably.

Example 4. An \( \text{SLID}_{\text{SNC}}[\text{dll}] \)-graph \( \mathcal{G} \) is given in Figure 2, where each node is just one variable. Labels of the arcs are abbreviated, e.g., the label \( \text{left}, E_1 \mapsto [(\text{left}, E_2), (\text{right}, E_3)] \) of the arc \((E_1, E_2)\) is abbreviated as \text{left}.

Our decision procedure for satisfiability follows [14] closely. The underpinning idea is, on a high level, to reduce the \( \text{SLID}_{\text{SNC}}[P] \) graph by exploiting that the subheaps represented by different spatial atoms must be domain disjoint. The main difference to [14] will be
discussed when they appear. We start with some notion of persistent set of arcs, largely from [14]. Intuitively, a set of arcs is persistent if in every model, a nonempty subheap has to be assigned to it.

**Definition 5 (Persistent set of arcs [14]).** Suppose $\mathcal{G} = (V, \mathcal{A}, \mathcal{N}, \mathcal{E}, \mathcal{L}, R_\phi)$ is an SLID$_{SNC}[P]$-graph and $\mathcal{E}' \subseteq \mathcal{E}$. Then $\mathcal{E}'$ is said to be persistent in $\mathcal{G}$ if either $\mathcal{E}'$ contains a field-labeled arc, or there are two nodes $[E], [F]$ in a connected component of $\mathcal{G}[\mathcal{E}']$ (the subgraph of $\mathcal{G}$ induced by $\mathcal{E}'$) such that $([E], [F]) \in R_\phi$.

For an SLID$_{SNC}[P]$-graph $\mathcal{G} = (V, \mathcal{A}, \mathcal{N}, \mathcal{E}, \mathcal{L}, R_\phi)$ and $e \in \mathcal{E}$, we use $\mathcal{G} - \text{atom}(e)$ to denote the graph obtained from $\mathcal{G}$ by removing all the arcs $e' \in \mathcal{E}$ such that $\text{atom}(e') = \text{atom}(e)$.

**Definition 6 (Reduced graphs).** An SLID$_{SNC}[P]$-graph $\mathcal{G} = (V, \mathcal{A}, \mathcal{N}, \mathcal{E}, \mathcal{L}, R_\phi)$ is reduced if $\mathcal{G} = \bot$, or otherwise if it satisfies the following conditions:

- (C1) For each $[E] \in \mathcal{N}$, $([E], [E]) \notin R_\phi$.
- (C2) For each arc $e$ out of $[E]$, $R_{\mathcal{G} - \text{atom}(e)}([E])$ is not persistent.
- (C3) For each arc $e$ out of $[E]$, $\text{nil}$ is not reachable from $[E]$ in $R_{\mathcal{G} - \text{atom}(e)}([E])$.
- (C4) For each pair of distinct nodes $[E], [F]$, there do not exist two predicate-labeled arc-disjoint simple paths from $[E]$ to $[F]$.

**Remark.** We would like to remark that the arcs $e$ in (C2) and (C3) may be field-labeled arcs. Moreover, we would like to mention the connection of the aforementioned conditions with those in [14, Table 2]: (C2) is adapted from conditions (i)—(iii) therein to deal with nonlinear structures, and (C4) is adapted from condition (iv) therein. On the other hand, (C3) is new to deal with the special variable nil ([14] did not consider nil).

**Example 7.** The graph $\mathcal{G}$ in Figure 2 is not reduced, because it violates (C2): For the arc $e_1 = (E_2, F_3)$, $R_{\mathcal{G} - \text{atom}(e_1)}(E_2)$ is persistent; (C3): For the arc $e_2 = (F_{11}, F_{12})$, nil is reachable from $F_{11}$ in $R_{\mathcal{G} - \text{atom}(e_2)}(F_{11})$; and (C4): There are two arc-disjoint predicate-labeled simple paths from $F_5$ to $F_8$. $\mathcal{G}$ can be transformed into a reduced graph, given as $\mathcal{G}_\varphi$ in Figure 3.

As in [14], deciding whether a non-$\bot$ SLID$_{SNC}[P]$ graph $\mathcal{G}$ is reduced can be done in polynomial time in the size of $\mathcal{G}$. To transform an arbitrary non-$\bot$ SLID$_{SNC}[P]$ graph in to a reduced one, we provide a procedure which checks, for a given input non-$\bot$ SLID$_{SNC}[P]$ graph

![Figure 2](image_url)
G if any condition in Definition 6 is violated. If this is the case, the algorithm performs certain operations (e.g., contract arcs, or merge nodes, or mark the graph to be ⊥ directly) depending on which condition is violated, until G is reduced. Most of these are similar to [14]. However, regarding the new condition (C3), if there is an arc e out of [E] and [null] is reachable from [E] in \( R_{\text{atom}}(\psi) ([E]) \), then we contract the arc e in G if e is predicate-labeled, and mark G to be ⊥ directly otherwise. The details of the procedure is omitted due to the page limit.

By Definition 6, when the graph G is reduced, there are only two possibilities: either G = ⊥ implying that the corresponding formula SLID_{SNC} \( [P] \) is unsatisfiable, or G ≠ ⊥ in which case one can construct a model out of G. The similar result for linked lists is rather straightforward [14]. However, it is a bit more involved in our setting, since we need to construct a nonlinear structure to witness the satisfiability of G.

**Theorem 9.** Suppose \( P \in \mathcal{P}_1 \). Then the satisfiability for SLID\( \text{SNC} \) \( [P] \) is in polynomial time.

### 4.2 Entailment

In this section, we consider the entailment problem, i.e., to decide whether \( \varphi \models \psi \) for two satisfiable SLID_{SNC} \( [P] \) formulae \( \varphi, \psi \) such that \( \text{Vars}(\psi) \subseteq \text{Vars}(\varphi) \). Our goal is to provide a polynomial time decision procedure for the entailment problem.

By Proposition 8, we assume that \( \mathcal{G}_\varphi = (\text{Vars}(\varphi), \text{Atoms}(\varphi), \mathcal{N}_\varphi, \mathcal{E}_\varphi, \mathcal{L}_\varphi, \mathcal{R}_{\varphi, \varphi}) \) and \( \mathcal{G}_\psi = (\text{Vars}(\psi), \text{Atoms}(\psi), \mathcal{N}_\psi, \mathcal{E}_\psi, \mathcal{L}_\psi, \mathcal{R}_{\psi, \psi}) \) are reduced. For \( E \in \text{Vars}(\varphi) \) (resp. \( E \in \text{Vars}(\psi) \)), we use \([E]_{\varphi}\) (resp. \([E]_{\psi}\)) to denote the node in \( \mathcal{G}_\varphi \) (resp. \( \mathcal{G}_\psi \)) that contains E. Moreover, we assume that for each \([E]_{\varphi}, [F]_{\varphi} \in \mathcal{N}_\varphi \) (resp. \([E]_{\psi}, [F]_{\psi} \in \mathcal{N}_\psi \)), if \( E = F \land \varphi \) (resp. \( E = F \land \psi \)) is unsatisfiable, then \([E], [F] \in \mathcal{R}_{\varphi, \varphi} \) (resp. \([E], [F] \in \mathcal{R}_{\psi, \psi} \)).

According to Proposition 8, these (possibly implicit) inequalities can be added to \( \mathcal{G}_\varphi \) and \( \mathcal{G}_\psi \) in polynomial time. Figure 3 depicts two graphs \( \mathcal{G}_\varphi \) and \( \mathcal{G}_\psi \), which will be used as the running example. \( \mathcal{G}_\varphi \) is obtained from the graph G in Example 4.

The main idea of our decision procedure is to utilise graph representations \( \mathcal{G}_\varphi \) and \( \mathcal{G}_\psi \), and to extend the concept of graph homomorphisms used in [14] to nonlinear structures and
We shall use examples to illustrate the main ideas of our decision procedure. Let us first

Definition 11

minimum subgraph of $G$ that matches $\varphi$. In addition, these simple paths are mutually arc-disjoint. The arcs that belong to these simple paths are called covered, and those that do not are called uncovered. As a result of the intuitionistic semantics, the uncovered arcs of $G$ can be ignored. The fact that the existence of a graph homomorphism from $G$ to some subgraph of $G$ can be decided in polynomial time is attributed to the unique simple path property enjoyed by $G$, that is, for each pair of distinct nodes, say $[E]_\varphi$ and $[F]_\varphi$, there is at most one simple path from $[E]_\varphi$ to $[F]_\varphi$. The unique simple path property of $G$ implies that the graph homomorphism from $G$ to some subgraph of $G$ is unique if it exists.

To deal with the entailment problem $\varphi \models_c \psi$ for SLID$_{SNC}[P]$ formulae $\varphi$ and $\psi$ in this paper, we need to generalise the concept of graph homomorphisms in [14] to nonlinear structures and deal with the classical semantics as follows:

- Since the structures defined by the predicate atoms in SLID$_{SNC}[P]$ are nonlinear in general, a graph homomorphism should map each predicate-labeled arc $e$ of $G$, to a potentially nonlinear subgraph $\text{Subgraph}_e(G)$ of $G$, instead of just a simple path. In addition, for each predicate-labeled arc $e$ of $G$, $\text{Subgraph}_e(G)$ should match $\text{atom}_\psi(e)$, that is, $\text{Subgraph}_e(G) \models \text{atom}_\psi(e)$. (Recall that we assume that an SLID$_{SNC}[P]$ formula and its graph representation are interchangeable.) Actually, we will choose $\text{Subgraph}_e(G)$ to be the minimum subgraph of $G$ that matches $\text{atom}_\psi(e)$, in order to guarantee the uniqueness of the graph homomorphism.

- Because we are considering the classical semantics in this paper, those uncovered arcs of $G$ cannot be simply ignored. We need to ensure that there is a way to dispatch the uncovered arcs of $G$ to the predicate-labeled arcs of $G$ so that for each predicate-labeled arc $e$ of $G$, $\text{atom}_\psi(e)$ is matched by the graph consisting of $\text{Subgraph}_e(G)$ and these uncovered arcs of $G$, which are dispatched to $e$.

- Finally, we need to show that checking whether a subgraph of $G$ matches $\text{atom}_\psi(e)$ for a predicate-labeled arc $e$ of $G$ can be done in polynomial time.

We shall use examples to illustrate the main ideas of our decision procedure. Let us first demonstrate, for each predicate-labeled arc $e$ of $G$, how we choose the graph $\text{Subgraph}_e(G)$ to be the minimum subgraph of $G$ that matches $\text{atom}_\psi(e)$. The following example shows that $G$ for an SLID$_{SNC}[P]$ formula $\varphi$ does not enjoy the unique simple path property in general.

Example 10. Let us consider the SLID$_{SNC}[P]$ formulae $\varphi, \psi$ whose graph representations are illustrated in Figure 3. The graph $G_\varphi$ does not satisfy the unique simple path property: there are two arc-disjoint simple paths from $E_0$ to $E_7$ in $G_\varphi$. Note that the subgraph of $G$ comprising all the arcs reachable from $E_0$ does match $\text{th}(E_0; E_7)$. This follows from the fact $E_0 \not\rightarrow ([\text{left}, F_{13}],[\text{right}, F_{14}]) \not\rightarrow \text{th}(F_{13}; E_7) \not\rightarrow \text{th}(F_{14}; \text{nil}) \not\rightarrow \text{th}(E_0; E_7)$ and $F_{14} \not\rightarrow ([\text{left}, E_7],[\text{right}, \text{nil}]) \not\rightarrow E_7 \not\rightarrow ([\text{left}, \text{nil}],[\text{right}, \text{nil}]) \not\rightarrow \text{th}(F_{14}, \text{nil})$.

Although the graph representations of SLID$_{SNC}[P]$ formulae do not satisfy the unique simple path property in general, we can still find a way to define $\text{Subgraph}_e(G_\varphi)$ as the minimum subgraph of $G_\varphi$ that matches $\text{atom}_\psi(e)$.

Definition 11 (Subgraph$_e(G_\varphi)$). Given a predicate-labeled arc $e = ([E]_\psi,[F]_\psi)$ in $G_\psi$, Subgraph$_e(G_\varphi)$ is defined as follows:
If $[E]_\varphi = [F]_\varphi$, then $\text{Subgraph}_G(G_\varphi)$ is the empty graph.

Otherwise,

- if $[F]_\varphi = [\text{nil}]_\varphi$, then $\text{Subgraph}_G(G_\varphi) = G_\varphi[\text{Range}_G([E]_\varphi)]$, that is, the subgraph of $G_\varphi$ comprising all the arcs reachable from $[E]_\varphi$,

- if $[F]_\varphi \neq [\text{nil}]_\varphi$,
  - if there is a unique simple path from $[E]_\varphi$ to $[F]_\varphi$, denoted by $\pi_e$, then $\text{Subgraph}_G(G_\varphi)$ is the subgraph of $G_\varphi$ comprising: (1) all the arcs in $\pi_e$, (2) all field-labeled arcs $e' = ([E']_\varphi, [F']_\varphi)$ such that $[E']_\varphi, [F']_\varphi \in \mathcal{N}(\pi_e)$ and $[F']_\varphi \neq [\text{nil}]_\varphi$, and (3) all arcs of $G_\varphi$ that are reachable from $[F']_\varphi$,
  - otherwise, $\text{Subgraph}_G(G_\varphi) = G_\varphi[\text{Range}_G([E]_\varphi)]$.

Example 12. Let us consider the formulae $\varphi, \psi$ illustrated in Figure 3. Let $e_1, e_2, e_3$ be the predicate-labeled arcs in $G_{\varphi}$, from $E_2$ to $E_3$, $E_3$ to $E_4$, and $E_6$ to $E_7$ respectively. Since $[E_2]_{\varphi} \neq [E_4]_{\varphi}, [E_4]_{\varphi} \neq [\text{nil}]_\varphi$, and there is a unique path from $[E_2]_{\varphi}$ to $[E_4]_{\varphi}$, according to Definition 11, $\text{Subgraph}_G(G_\varphi)$ comprises the unique simple path from $[E_2]_{\varphi}$ to $[E_4]_{\varphi}$, the arc from $[E_2]_{\varphi}$ to $[E_3]_{\varphi}$, and all the arcs reachable from $[E_3]_{\varphi}$. Similarly, $\text{Subgraph}_G(G_\varphi)$ comprises the unique simple path from $[E_3]_{\varphi}$ to $[E_4]_{\varphi}$ in $G_\varphi$, which is just the predicate-labeled arc from $[E_3]_{\varphi}$ to $[E_4]_{\varphi}$. Since $[E_6]_{\varphi} \neq [E_7]_{\varphi}, [E_7]_{\varphi} \neq [\text{nil}]_\varphi$, and there are two distinct simple paths from $[E_6]_{\varphi}$ to $[E_7]_{\varphi}$, we have that $\text{Subgraph}_G(G_\varphi) = G_\varphi[\text{Range}_G([E_6]_{\varphi})]$, that is, the subgraph of $G_\varphi$ comprising all the arcs reachable from $[E_6]_{\varphi}$.

Example 13. Let $\varphi = \text{lseg}(E_1; E_2) \land \text{lseg}(E_2; E_3) \land \text{lseg}(E_3; E_4) \land \text{lseg}(E_3; E_4)$, $\psi = \text{lseg}(E_1; E_4)$, and $G_\varphi$ and $G_\psi$ are depicted in Figure 4. (For simplicity, arc label ($\text{lseg}$, $\text{lseg}(E_1; E_2)$ is abbreviated as $\text{lseg}$, sic passim.) We claim $\varphi \models \psi$. To see this, let $(s, h)$ be the state such that $s(E_1) = 1, s(E_2) = 2, s(E_3) = 3, s(E_4) = 3, h(1, \text{next}) = 2, h(2, \text{next}) = 3, h(3, \text{next}) = 2$. Then $(s, h) \models \varphi$, but $(s, h) \not\models \psi$ under the classical semantics, although the subheap $h'$ of $h$ with $\text{Idom}(h') = \{1, 2\}$ satisfies $\psi$. This is in contrast to the intuitionistic semantics, under which $h \models \psi$ follows from $h' \models \psi$.

Regarding the homomorphism [14], although the unique simple path from $E_1$ to $E_4$ in $G_\varphi$ matches $\text{lseg}(E_1; E_4)$, the arc $(E_3, E_2)$ in $G_\varphi$ is uncovered. This is the issue of using graph homomorphism to entailment under the classical semantics, as also pointed out in [14]. On
the other hand, let us assume another pair of graphs in Figure 5. It is not hard to see that the entailed \( \varphi \models \psi \) holds. Interestingly, in this case, all the arcs in the cycle \( E_2 \cong E_3 \) in \( \mathcal{G}_\varphi \) are uncovered.

As suggested in Example 13, to deal with the classical semantics, it is necessary to put some proper constraints on the uncovered arcs of nontrivial SCCs in \( \mathcal{G}_\varphi \). Let us introduce some additional notations for \( \mathcal{G}_\varphi \). An arc \( e' \in \mathcal{G}_\varphi \) is covered if

1. there is a covered arc whose destination node belongs to cycles, where all \( \varphi \) satisfy some additional constraint given below.
2. or \( e' \) belongs to \( \text{Subgraph}_e(\mathcal{G}_\varphi) \) for some predicate-labeled arc \( e \in \mathcal{G}_\psi \).

A simple cycle \( C \) of \( \mathcal{G}_\varphi \) is covered if all arcs in \( C \) are covered; \( C \) is uncovered if none of the arcs in \( C \) are covered. A nontrivial SCC \( S \) is said to be final w.r.t. covered arcs if there are no covered arcs that are reachable from the nodes in \( S \), moreover, one of the following holds: 1) there is a covered arc whose destination node belongs to \( S \), 2) there is a predicate-labeled arc \( e \) from \( [E]_{\varphi} \) to \( [F]_{\varphi} \) in \( \mathcal{G}_\psi \) such that \( [E]_{\varphi} = [F]_{\varphi} \) is in \( S \). A collection of simple cycles \( S = \{C_1, \ldots, C_n\} \) is said to be unentangled if \( C_i \) and \( C_j \) (\( i \neq j \)) do not share any arc, and share at most one node.

In the sequel, we first analyse the structure of nontrivial SCCs in \( \mathcal{G}_\varphi \).

**Proposition 14.** Suppose \( \varphi, \psi \) are two \( \text{SLID}_\text{SNC}[P] \) formulae such that \( \varphi \models \psi \). Then the nontrivial SCCs of \( \mathcal{G}_\varphi \) satisfy the following structural constraints.

- If a nontrivial SCC \( S \) of \( \mathcal{G}_\varphi \) contains only predicate-labeled arcs, then \( S \) comprises a collection of unentangled simple cycles \( \{C_1, \ldots, C_n\} \).
- If a nontrivial SCC \( S \) of \( \mathcal{G}_\varphi \) contains both an uncovered arc and a field-labeled arc, then \( S \) is final w.r.t. covered arcs and is exactly a simple cycle.

A graph homomorphism from \( \mathcal{G}_\psi \) to \( \mathcal{G}_\varphi \) is required to satisfy that each simple cycle \( C \) of \( \mathcal{G}_\varphi \) is either covered or uncovered. In addition, the uncovered simple cycles of \( \mathcal{G}_\varphi \) should satisfy some additional constraint given below.

**Definition 15.** Let \( S = \{C_1, \ldots, C_n\} \) be a nontrivial SCC of \( \mathcal{G}_\varphi \) with unentangled simple cycles, where all \( C_i \)’s are either covered or uncovered, with at least one uncovered \( C_i \).

- An uncovered connected component (UCC) \( C \) of \( S \) is a connected component of the graph obtained from \( S \) by removing all covered simple cycles.
- Suppose \( S \) is not final w.r.t. covered arcs. Then \( S \) is indirectly covered if for each UCC \( C \) of \( S \) and each simple cycle \( C \in C \), and there is an arc \( e = ([E]_{\varphi}, [F]_{\varphi}) \) in \( \mathcal{G}_\psi \) with \( \text{atom}_{\varphi}(e) = P(E; F; B) \), satisfying one of the following constraints:
  1. \( [E]_{\varphi} = [F]_{\varphi} \in C \), and for each arc \( e' \in C \) with \( \text{atom}_{\varphi}(e') = P(E'; F'; B') \), it holds that \( B \sim_{\varphi} B' \) (where \( \sim_{\varphi} \) is extended to vectors of variables in an obvious way).
  2. \( [E]_{\varphi} \neq [F]_{\varphi} \), there is a node belonging to both \( \text{Subgraph}_e(\mathcal{G}_\varphi) \) and \( C \), and for each arc \( e' \in C \) with \( \text{atom}_{\varphi}(e') = P(E'; F'; B') \), it holds that \( B \sim_{\varphi} B' \).

Note that if the predicate \( P \) contains no static parameters, then the condition is simplified as follows: for each UCC \( C \) of \( S \), there is an arc \( e = ([E]_{\varphi}, [F]_{\varphi}) \) in \( \mathcal{G}_\psi \), and let \( [E]_{\varphi} = [F]_{\varphi} \in C \) or there is a node belonging to both \( \text{Subgraph}_e(\mathcal{G}_\varphi) \) and \( C \).

- Suppose \( S \) is final w.r.t. covered arcs. Then \( S \) is indirectly covered if one of the following holds: 1) there is a covered arc \( e' \in \mathcal{G}_\varphi \) with \( \text{atom}_{\varphi}(e') = P(E; F; B) \) such that \( [F]_{\varphi} \) is in \( S \) and \( \mathcal{G}_\varphi[\mathcal{R}_{\mathcal{G}_\varphi}([F]_{\varphi})] \) matches \( P(F; F; B) \), 2) there is an arc \( e \in \mathcal{G}_\psi \) with \( \text{atom}_{\varphi}(e) = P(E; F; B) \) such that \( [E]_{\varphi} = [F]_{\varphi} \) is in \( S \) and \( \mathcal{G}_\varphi[\mathcal{R}_{\mathcal{G}_\varphi}([F]_{\varphi})] \) matches \( P(F; F; B) \).
Tractability of Separation Logic with Inductive Definitions: Beyond Lists

Definition 16 (Criteria of coverage of arcs). A graph homomorphism from $\mathcal{G}_\psi$ to $\mathcal{G}_\varphi$ is fully covered, if

(I) for each nontrivial SCC $S$ that is not final w.r.t. covered arcs and contains an uncovered arc (which implies that $S$ contains predicate-labeled arcs only by Proposition 14), each simple cycle $C$ in $S$ is either covered or uncovered, and each UCC of $S$ is indirectly covered,

(II) for each nontrivial SCC $S$ that is final w.r.t. covered arcs, $S$ is indirectly covered,

(III) each uncovered arc of $\mathcal{G}_\varphi$ either belongs to some nontrivial SCC or is reachable from a node in some nontrivial SCC that is final w.r.t. covered arcs.

We now collate all the ingredients together and specify the formal definition of the graph homomorphism from $\mathcal{G}_\psi$ to $\mathcal{G}_\varphi$.

Definition 17 (Graph homomorphism from $\mathcal{G}_\psi$ to $\mathcal{G}_\varphi$). A homomorphism from $\mathcal{G}_\psi$ to $\mathcal{G}_\varphi$ is a function $\theta : N_\psi \rightarrow N_\varphi$ satisfying the following constraints.

Variable subsumption: For each $E \in \text{Vars}(\psi)$, $[E]_\psi \subseteq \theta([E]_\varphi)$. In particular, we have $E \in \theta([E]_\varphi)$. This implies that $\theta([E]_\varphi)$, and thus, for convenience, we sometime use $[E]_\varphi$ to denote $\theta([E]_\psi)$.

Field-labeled arc: For each field-labeled arc $e = ([E]_\psi, [F]_\psi)$ in $\mathcal{G}_\psi$ with $L_\psi(e) = (f, E \mapsto \rho)$, there is a field-labeled arc $e' = ([E]_\varphi, [F]_\varphi)$ in $\mathcal{G}_\varphi$ with $L_\varphi(e') = (f', E' \mapsto \rho')$, and for each $(f', F_1)$ in $\rho$ and $(f', F_2)$ in $\rho'$, $\theta([F_1]_\psi) = [F_1]_\varphi$.

Predicate-labeled arc: For each predicate-labeled arc $e = ([E]_\psi, [F]_\psi)$ in $\mathcal{G}_\psi$ with $\text{atom}_\psi(e) = P(E; F; B)$ and $[F]_\varphi \neq [F]_\psi$, $\text{Subgraph}_t(\mathcal{G}_\psi)$ matches $P(E; F; B)$.

Inequality: For each $([E]_\psi, [F]_\psi) \in R_{\varphi, \psi}$, $([E]_\varphi, [F]_\varphi) \in R_{\varphi, \psi}$.

Coverage: $\theta$ is fully covered (cf. Definition 16).

Separation constraint:

- For each pair of distinct arcs $e_1, e_2$ in $\mathcal{G}_\psi$, the two subgraphs $\text{Subgraph}_{t_1}(\mathcal{G}_\varphi)$ and $\text{Subgraph}_{t_2}(\mathcal{G}_\varphi)$ are arc-disjoint.
- For every two distinct nontrivial SCCs $S_1, S_2$ of $\mathcal{G}_\varphi$ that are final w.r.t. covered arcs, the sets of arcs that are reachable from (nodes in) $S_1$ and $S_2$ respectively are disjoint.

Proposition 18. $\varphi \models_c \psi$ iff there is a graph homomorphism from $\mathcal{G}_\psi$ to $\mathcal{G}_\varphi$.

Example 19. Consider the graphs $\mathcal{G}_\varphi, \mathcal{G}_\psi$ in Figure 3. The function $\theta : N_\psi \rightarrow N_\varphi$ can be defined obviously, for instance $\theta(E_2) = \{E_2, F_4\}$ in $\mathcal{G}_\psi$, it is a routine to check that $\text{Subgraph}_t(\mathcal{G}_\varphi)$ matches $e$. Similarly for the other predicate-labeled arcs in $\mathcal{G}_\psi$. In addition, the homomorphism $\theta$ is fully covered. Let $S_1, S_2$ be the uncovered nontrivial SCCs in $\mathcal{G}_\varphi$, which contain the node $\{F_3, F_7, F_8\}$ and $E_5$ respectively. $S_1$ is not final w.r.t. covered arcs and contains only one uncovered connected component (i.e., itself). Since $tb$ contains no static parameters and the arc from $F_1$ to $\{F_3, F_7, F_8\}$ is covered, according to Definition 15, we know that $S_1$ is indirectly covered. Moreover, $S_2$ is final w.r.t. covered arcs. Because the arc from $E_3$ to $E_5$ is covered and $\mathcal{G}_\varphi[R_{\mathcal{G}_\varphi}(E_5)]$, which comprises all the arcs reachable from $E_5$ in $\mathcal{G}_\varphi$, matches $tb(E_5; E_5)$, we deduce that $S_2$ is indirectly covered as well. Finally, it is easy to see that the separation constraint is satisfied. Therefore, $\theta$ is a graph homomorphism, and we conclude that $\varphi \models_c \psi$.

Finally, in order to show that the existence of a graph homomorphism from $\mathcal{G}_\varphi$ to $\mathcal{G}_\psi$ can be decided in polynomial time, it remains to show that checking that a subgraph of $\mathcal{G}_\varphi$ matches a predicate atom $P(E; F; B)$ can be decided in polygonal time.

Proposition 20. Given a subgraph $\mathcal{G}$ of $\mathcal{G}_\varphi$ and a predicate atom $P(E; F; B)$, one can decide whether $\mathcal{G}$ matches $P(E; F; B)$ in polynomial time.
The proof of Proposition 20 relies on the fact that the problem of whether $G$ matches $P(E; F; B)$ can be reduced to checking some structural properties of $G$, which can be done in polynomial time.

Combining Proposition 18 and Proposition 20, we conclude:

\textbf{Theorem 21.} Let $P \in \mathcal{P}_1$. The entailment problem for $\text{SLID}_{\text{SNC}}[P]$ is in polynomial time.

\textbf{A detailed comparison with [16].}

Our polynomial-time decision procedure for $\text{SLID}_{\text{SNC}}[P]$ formulae with $P \in \mathcal{P}_1$ is related to that in [16], since the parameters of an inductive predicate in [16] are also divided as source, destination, and static parameters. Nevertheless, our decision procedure differs from that in [16] in the following three aspects. (1) $\text{SLID}_{\text{SNC}}[P]$ formulae with $P \in \mathcal{P}_1$ and the fragment of SL in [16] are incomparable: The latter allows nesting inductive predicates so that nested lists and skip lists can be defined, while our fragment disallows this. On the other hand, $\text{SLID}_{\text{SNC}}[P]$ formulae with $P \in \mathcal{P}_1$ allows defining tree structures, which are not supported in [16]. (2) The inequality $E \not= F$ may appear in the inductive rules of predicates in [16], but is excluded here. As a result, the fragment in [16] is \textit{precise} in the sense that when checking $\varphi \models \psi$, for each arc $e$ in $G_\varphi$, there is a unique subgraph of $G_\psi$ that matches $\text{atom}_\varphi(e)$. On the other hand, $\text{SLID}_{\text{SNC}}[P]$ is not precise because of the uncovered simple cycles of $G_\varphi$. Some proper constraints on these uncovered simple cycles are necessary in the graph homomorphism to achieve a \textit{complete} decision procedure for the entailment problem. (3) In [16], checking whether a subgraph of $G_\varphi$ matches a predicate atom $P(E; F; B)$ is reduced to the membership problem of a tree automaton constructed from the inductive definitions of $P$, which is of exponential size in the worst case, due to the nested inductive predicates. However, in our decision procedure, we simply check whether the subgraph satisfies some structural properties in polynomial time.

\section{Conclusion}

We have provided polynomial-time satisfiability and entailment checking algorithms for $\text{SLID}_{\text{SNC}}$ with one source (destination) parameter. We have also shown that the satisfiability problem is generally \textit{NP}-complete if more than one source (destination) parameter is allowed, particularly for an inductive predicate of doubly linked list segments.

For the future work, we strongly believe that our polynomial-time decision procedures can be further generalised to handle inductive predicates with multiple source (destination) parameters in a constrained form. Moreover, adding data constraints certainly is one of the promising extensions of the current work.

\textbf{References}

Tractability of Separation Logic with Inductive Definitions: Beyond Lists


Abstract

We extend the classical model of multi-pushdown systems by considering systems that operate on a finite set of variables ranging over natural numbers. The conditions on variables are defined via gap-order constraints that allow to compare variables for equality, or to check that the gap between the values of two variables exceeds a given natural number. Furthermore, each message inside a stack is equipped with a data item representing its value. When a message is pushed to the stack, its value may be defined by a variable. When a message is popped, its value may be copied to a variable. Thus, we obtain a system that is infinite in multiple dimensions, namely we have a number of stacks that may contain an unbounded number of messages each of which is equipped with a natural number. It is well-known that the verification of any non-trivial property of multi-pushdown systems is undecidable, even for two stacks and for a finite data-domain. In this paper, we show the decidability of the reachability problem for the classes of data multi-pushdown system that admit a bounded split-width (or equivalently a bounded tree-width). As an immediate consequence, we obtain decidability for several subclasses of data multi-pushdown systems. These include systems with single stacks, restricted ordering policies on stack operations, bounded scope, bounded phase, and bounded context switches.

1998 ACM Subject Classification D.2.4 Software/Program Verification

Keywords and phrases Pushdown Systems, Model-Checking, Gap-Order, Bounded Split-Width

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2017.38

1 Introduction

In the last few years, a lot of efforts have been devoted to the verification of discrete program models that have infinite state spaces such as Petri nets, lossy channel machines and multi-pushdown systems. In particular, multi-pushdown systems have been extensively studied as a natural model for concurrent Boolean recursive programs. Unfortunately, multi-pushdown systems are in general Turing powerful, and hence all basic decision problems are undecidable for them [31]. To overcome the undecidability barrier, several subclasses of multi-pushdown systems have been proposed (e.g., [16, 30, 10, 2, 25, 29, 26, 34, 11, 13, 12, 24, 6, 5, 33, 28]).

Bounded context-switch has been proposed in [30] as an adequate criterion for the verification of multi-pushdown systems. The idea is to restrict the analysis to executions that can be split into a given number of contexts where, in each context, pop and push operations are exclusive to one stack. The context-bounded reachability problem is NP-complete, though the state space that can be explored is still unbounded.

© Parosh Aziz Abdulla, C. Aiswarya, and Mohamed Faouzi Atig; licensed under Creative Commons License CC-BY

28th International Conference on Concurrency Theory (CONCUR 2017).
Editors: Roland Meyer and Uwe Nestmann; Article No. 38; pp. 38:1–38:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany
In [25], La Torre et al. generalize the notion of context into phase. A phase is a sequence of operations in which at most one stack can be popped, while there is no restriction on the push operations. The bounded-phase restriction considers only executions of the system that can be split into a given number of phases. In this case, the phase-bounded reachability problem is decidable in double exponential time.

Another generalization of bounded context-switch is bounded scope [26] which restricts the analysis of the multi-pushdown system to those executions in which the number of context-switches between any push operation and its corresponding pop operation is bounded by a given number. This definition extends the notion of contexts in term of coverage while being orthogonal to the notion of phase. In [26], the scope-bounded reachability problem is shown to be PSPACE-complete.

Another way to obtain decidability is to impose a linear order on stacks [16]. Stack operations are constrained in such a way that any pop operation is only allowed on the first non-empty stack. In [10], the reachability problem is shown to be 2ETIME-complete when assuming this ordering policy on stack operations. Furthermore, imposing such a restriction strictly extends the notion of phases while being orthogonal to scope-boundedness.

In [29, 21, 27], a unified technique to reason about multi-pushdown systems under such restrictions is presented. The idea is to see an execution as a graph with extra edges relating push operations and their corresponding pop operations. Then, the authors prove that the graphs generated under these restrictions have bounded split-width (or equivalently bounded tree-width). As an immediate consequence of Courcelle’s theorem [20], the decidability of the reachability problem for multi-pushdown systems under these restrictions is obtained.

However, all these models assume a finite-state control, which means that the variables of the modelled programs are assumed to range over finite domains. Several extensions of (multi-)pushdown systems with data have been studied in the literature (see e.g., [14, 8, 1, 17, 22]). Most of these extensions concern the case of multi-pushdown systems with one stack except the work presented in [14] where an extension of multi-pushdown systems with data has been proposed. In order to obtain decidability of the reachability problem, the model requires the strong assumption of data freshness, and the restriction of the stack accesses to the bounded phase policy. Furthermore, the variable operations are restricted to checking (dis)equality.

In this paper, we consider an extension of multi-pushdown systems, which we call Data Multi-Push-Down Automata (DMPDA), that strengthens the classical model in two ways. First in addition to stacks, a DMPDA uses a finite set of variables ranging over the natural numbers. Moreover, each message inside the stack is equipped with a natural number which represents its value. Thus, we obtain a model that is possibly unbounded in multiple dimensions, namely we have a number of stacks such that each stack may contain an unbounded number of messages each of which is equipped with a natural number. The operations allowed on variables are defined by the gap-order constraint system [18, 32]. More precisely, DMPDA allow to compare the values of variables for equality, or to check that the gap between the values of two variables exceeds a given natural number. Also, a variable may be assigned a new arbitrary value, the value of another variable, or a value that is larger than at least a given natural number than the current value of another variable. Furthermore, a push operation may copy the value of a variable to the pushed message, and a pop operation may copy the value attached to the popped message to a variable. In this manner, the model of DMPDA subsumes two basic models, namely multi-pushdown systems (that we get by removing the variables and neglecting the values associated to the pushed messages) and the model of integral relational automata [18] (that we get by removing all the stacks).

Our main result is the decidability of the reachability problem for the classes of DMPDA.
that admit a bounded split-width. To that aim, we solve a more general problem, namely we characterize the reachability relation on variables between each pair of control states. More precisely, we present an algorithm for computing a finite set of gap-order formulas whose denotations describe values of variables that allow to reach one state from another with empty stacks. The main ingredient of the algorithm is a symbolic representation, called traces, that encode certain transition sequences in the automaton. A trace represents a set of partial runs. A partial run does not record the contents of the stacks, but marks positions inside the run that correspond to matching push/pop operations. Furthermore, a partial run is not contiguous in the sense that it may contain a number of “holes”. Our algorithm will characterize the relation between the variables at the points where the holes occur. In particular, a partial run with no holes corresponds to a concrete run that starts and ends with empty stacks. The definition of partial runs allows to extend naturally the notion of split width [21] that has been considered for the analysis of multi-pushdown systems (without data). Intuitively, a run has a bounded split width if it can be built from atomic runs by using a shuffle and a contraction operator without producing any intermediate runs with more holes than the given bound. An atomic run is one that consists either of a single transition, or a pair of matching push/pop transitions. We show that our algorithm is guaranteed to terminate for all classes of systems that generate runs with a bounded split width. As an immediate consequence, we obtain the decidability for several subclasses of multi-pushdown systems with data including the ones that restrict the ordering policy on stack operations, or bound the scope, the number of phases, or the number of context switches.

**Related work.** Several subclasses of multi-pushdown systems have been proposed in the literature including bounded-context [30], bounded-phase [25], bounded scope [26] and ordered multi-pushdown systems [16]. The reachability problem for these classes has been shown to be decidable under the assumption of finiteness of the set of control states. These classes are subclasses of our model DMPDA and our decidability result subsumes the decidability of the reachability problem for these models. In contrast, we do not provide any complexity results.

Split-width and tree-width\(^1\) have been used for showing, in a unified way, the MSO decidability of several classes of multi-pushdown systems [29, 21, 27]. The method has been extended for message passing systems[6] and parameterized message passing systems[23]. However the considered models are restricted to the manipulation of variables over finite data domains while in our model, variables range over natural numbers. In fact the results presented in [29, 21, 27] are orthogonal to our result since we do not consider the model-checking problem against monadic second order logic.

Decidability of the reachability problem for pushdown systems (i.e., multi-pushdown systems with one stack) with data has been extensively studied in the literature (see e.g., [1, 17, 22, 3, 15, 19]). The closest work is pushdown systems with gap-order constraints [1], which is subsumed by our model. Furthermore, the techniques used to show the decidability of the reachability problem for pushdown systems with gap-order constrains are different from the ones used in this paper.

Extensions of multi-pushdown systems with data have been studied in [14]. This work uses the strong assumption of freshness of data, and bounded phase restriction on stack accesses. In contrast, we do not assume the freshness of data and our results can be applied to several subclass of multi-pushdown systems.

\(^1\) Split-width and tree-width are not identical, but one is bounded if and only if the other is. Further the bounds are related linearly [21].
In [8] the split-width technique is lifted to analyze timed multi-pushdown systems. Timed systems give rise to an infinite data domain. However, reachability in this case can be reduced to MSO model checking of untimed systems with finite propositional labelling indicating timing constraints, since realizability of a word with timing constraints can be expressed in MSO [7]. The crux of the decidability proof in all these cases is the use of tree-automata. In contrast, the reachability problem of DMPDA under bounded split-width does not reduce to the Boolean case. Furthermore, our algorithm uses a fix-point computation which terminates, thanks to well quasi-ordering of gap-order formulas.

2 Preliminaries

Let \( \mathbb{N} \) denote the set of natural numbers. For sets \( A \) and \( B \), we use \( f : A \to B \) to denote that \( f \) is a function from \( A \) to \( B \). We use \( f[a \mapsto a'] \) to denote the function \( f' \) such that \( f'(a) = a' \), and \( f'(x) = f(x) \) if \( x \neq a \). For \( A' \subseteq A \), we use \( f \circ A' \) to denote the restriction of \( f \) to \( A' \). For sets \( A_1 \) and \( A_2 \) with \( A_1 \cap A_2 = \emptyset \), and functions \( f_1 : A_1 \to B \) and \( f_2 : A_2 \to B \), we use \( f_1 \cup f_2 : A_1 \cup A_2 \to B \) to denote the function \( g \) such that \( g(a) = f_1(a) \) if \( a \in A_1 \) and \( g(a) = f_2(a) \) if \( a \in A_2 \). For a finite set \( A \), we use \( |A| \) to denote the size of \( A \).

For a set \( A \), we use \( A^* \) to denote the set of finite words over \( A \). We use \( \epsilon \) to denote the empty word. For \( w_1, w_2 \in A^* \), we use \( w_1 \cdot w_2 \) to denote the concatenation of \( w_1 \) and \( w_2 \).

Consider a set \( A \) and a total ordering \( \leq \) on \( A \). We use \( a_1 < a_2 \) to denote that \( a_1 \leq a_2 \) and \( a_1 \neq a_2 \). We use \( < \) to denote the induced immediate successor relation, i.e., \( a_1 < a_2 \) iff \( a_1 < a_2 \) and there is no \( a_3 \) such that \( a_1 < a_3 < a_2 \).

3 Model

In this section, we introduce Data Multi-Pushdown Automata (DMPDA). A DMPDA operates on multiple unbounded stacks each of which allows pushing (appending) and popping (removing) messages in a last-in-first-out manner. In addition to the stacks, a DMPDA uses a finite set of variables ranging over the natural numbers.

The allowed operations on variables are defined by the gap-order constraint system [18, 32]. More precisely, the model allows non-deterministic value assignment, copying the value of one variable to another, and assignment of a value \( v \) to some variable such that \( v \) is larger than at least a given natural number than the current value of another variable. The transitions may be conditioned by tests that compare the values of two variables for equality, or that give the smallest allowed gap between two variables. In addition to carrying a name (taken from a finite alphabet), each message inside a stack is equipped by a natural number that represents its “value”. A push operation may copy the value of a variable to the pushed message, and a pop operation may copy the value of the popped message to a variable. Notice that DMPDA extend the classical model of Push-Down Automata in three ways, namely they allow

(i) multiple stacks,
(ii) numerical variables, and
(iii) an infinite (numerical) stack alphabet.

Syntax

In the rest of the paper, we assume a finite set of variables \( \mathcal{X} \), a finite set of stacks \( \Sigma \), and a finite stack alphabet \( \Gamma \). A DMPDA \( A \) is a tuple \((Q, \Delta)\) where \( Q \) is a finite set of states, and \( \Delta \) is a finite set of transitions. A transition \( t \in \Delta \) is a triple \((q_1, op, q_2)\) where \( q_1, q_2 \in Q \) are states, and \( op \) is an operation of one of the following forms:
We define the Dmpda $\Delta$ for each stack $q$ in the form:

\[ \Delta = (\text{error}, \alpha, \beta) \]

where $\alpha : \Sigma \to \mathbb{N}$ defines the values of the variables, and $\beta : \Gamma \to (\Gamma \times \mathbb{N})^*$ defines, for each stack $\sigma \in \Sigma$, its content $\beta(\sigma)$. The content of a stack is a word whose elements are of the form $(a, c)$ where $a$ is a symbol and $c$ is its value. In particular, we define $\beta_\epsilon$ such that $\beta_\epsilon(\sigma) = \epsilon$ for all $\sigma \in \Sigma$. We say that $c$ is plain if $\beta = \beta_\epsilon$.

**Semantics**

A Dmpda induces a transition system as follows. A configuration $c$ is a triple $(q, \alpha, \beta)$ where $q \in Q$ is a state, $\alpha : \Sigma \to \mathbb{N}$ defines the values of the variables, and $\beta : \Sigma \to (\Gamma \times \mathbb{N})^*$ defines, for each stack $\sigma \in \Sigma$, its content $\beta(\sigma)$. The content of a stack is a word whose elements are of the form $(a, c)$ where $a$ is a symbol and $c$ is its value. In particular, we define $\beta_\epsilon$ such that $\beta_\epsilon(\sigma) = \epsilon$ for all $\sigma \in \Sigma$. We say that $c$ is plain if $\beta = \beta_\epsilon$.

**Inference rules defining the relation $\xrightarrow{t}$, where $t = (q, op, q')$:**

<table>
<thead>
<tr>
<th>Rule</th>
<th>Inference</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$nop$</td>
<td>$(q, \alpha, \beta) \xrightarrow{t} (q', \alpha, \beta)$</td>
<td>$nop$ is the empty operation that does not change the values of the variables or the contents of the stacks.</td>
</tr>
<tr>
<td>$x \leftarrow *$</td>
<td>$(q, \alpha, \beta) \xrightarrow{t} (q', \alpha \leftarrow c, \beta)$</td>
<td>$x \leftarrow *$ assigns non-deterministically an arbitrary value in $\mathbb{N}$ to the variable $x$.</td>
</tr>
<tr>
<td>$x \leftarrow y$</td>
<td>$(q, \alpha, \beta) \xrightarrow{t} (q', \alpha \leftarrow (y), \beta)$</td>
<td>$x \leftarrow y$ copies the value of variable $y$ to $x$.</td>
</tr>
<tr>
<td>$(x \leftarrow (&gt; c) y)$</td>
<td>$(q, \alpha, \beta) \xrightarrow{t} (q', \alpha \leftarrow (x \leftarrow (y)), \beta)$</td>
<td>$x \leftarrow (&gt; c) y$ assigns non-deterministically to $x$ a value that exceeds the current value of $y$ by $c$ (so the new value of $x$ is $&gt; y + c$).</td>
</tr>
<tr>
<td>$x = y$</td>
<td>$(q, \alpha, \beta) \xrightarrow{t} (q', \alpha \leftarrow (x \leftarrow (y)), \beta)$</td>
<td>$x = y$ checks whether the value of $x$ is equal to the value of $y$.</td>
</tr>
<tr>
<td>$x &lt; y$</td>
<td>$(q, \alpha, \beta) \xrightarrow{t} (q', \alpha \leftarrow (x \leftarrow (y)), \beta)$</td>
<td>$x &lt; y$ checks whether the gap between the values of $y$ and $x$ is larger than $c$.</td>
</tr>
<tr>
<td>$push(\sigma)(a)(x)$</td>
<td>$(q, \alpha, \beta) \xrightarrow{t} (q', \alpha \leftarrow (a \leftarrow (\sigma \cdot (\sigma))), \beta)$</td>
<td>$push(\sigma)(a)(x)$ pushes the symbol $a \in \Gamma$ to the stack $\sigma \in \Sigma$ and assigns to it the value of the variable $x$.</td>
</tr>
<tr>
<td>$pop(x)(\sigma)(a)$</td>
<td>$(q, \alpha, \beta) \xrightarrow{t} (q', \alpha \leftarrow (a \leftarrow (\sigma \cdot (\sigma))), \beta)$</td>
<td>$pop(x)(\sigma)(a)$ pops the symbol $a \in \Gamma$ (if $a$ is the top-most symbol at the stack $\sigma \in \Sigma$) and assigns its value to the variable $x$.</td>
</tr>
</tbody>
</table>

We define the source $src(t) = q_1$ and the target $tgt(t) := q_2$.
We define the transition relation $\rightarrow := \cup_{t \in \Delta} t^*$ on the set of configurations, where $t^*$ describes the effect of the transition $t$. The semantics of the transition relation is presented through the inference rules of Fig. 1, explained below one by one.

- $nop$. The values of the variables and the stack contents are not changed.
- $x \leftarrow *$. The value of the variable $x$ is changed non-deterministically to some natural number. The values of the other variables and the stack contents are not changed.
- $x \leftarrow y$. The value of the variable $y$ is copied to the variable $x$. The values of the other variables and the stack contents are not changed.
- $x \leftarrow (\succ_{c} y)$. The variable $x$ is assigned non-deterministically a value that exceeds the value of $y$ by $c$. The values of the other variables and the stack contents are not changed.
- $x = y$ (resp. $x <_{c} y$). The transition is only enabled if the value of $y$ is equal to the value of $x$ (resp. larger than the value of $x$ by more than $c$). The values of the variables and the stack contents are not changed.

- $push(\sigma)(a)(x)$. The symbol $a$ is pushed onto the stack $\sigma$ with a value equal to that of $x$.
- $pop(x)(\sigma)(a)$. The symbol $a$ is popped from the stack $\sigma$ (if it is the top-most symbol of $\sigma$), and its value is copied to the variable $x$.

We use $\rightarrow^*$ to denote the reflexive transitive closure of $\rightarrow$. A run $\pi$ is an alternating sequence $c_0; c_1; \ldots; c_{n-1}; c_n$ of configurations and transitions such that $c_i \xrightarrow{t_i} c_{i+1}$ for all $1 \leq i \leq n$. We say that $\pi$ is plain if $c_0$ and $c_n$ are plain. For configurations $c$ and $c'$, we write $c \xrightarrow{\pi} c'$ to denote that there is a run $\pi$ of the above form such that $c_0 = c$ and $c_n = c'$. Notice that $c \xrightarrow{\pi} c'$ iff $c \xrightarrow{\pi^*} c'$ for some run $\pi$.

### Reachability

In the Reachability Problem, we are given two plain configurations $c_1$ and $c_2$, and are asked whether $c_1 \xrightarrow{\pi} c_2$. In order to solve the reachability problem we will study reachability relations. For states $q,q' \in Q$, we define $R(q,q') := \{ (\alpha,\alpha') | q,\alpha,\beta \xrightarrow{\pi} q',\alpha',\beta' \}$. Intuitively, we summarize the values of the variables that allow us to move from $q$ to $q'$, starting and ending with empty stacks. More precisely, $R(q,q')$ contains all pairs $(\alpha,\alpha')$ such that we can start from a configuration where the state is $q$, the values of the variables are given by $\alpha$, and the stacks are empty to another configurations where the state is $q'$, the values of the variables are given by $\alpha'$, and the stacks are empty again.

### 4 Gap-Order Formulas

Fix a set $X$ of variables ranging over $\mathbb{N}$. An atomic gap-order formula over $X$ is either of the form $x = y$ or of the form $x <_{c} y$ where $x,y \in X$ and $c \in \mathbb{N}$. A gap-order formula $\phi$ over $X$ is a conjunction of atomic constraints over $X$. Sometimes, we represent $\phi$ as a set (containing all its conjuncts). For a function $Val : X \rightarrow \mathbb{N}$, we write $(as expected) Val = \phi$ to denote that $Val$ satisfies $\phi$. We will also consider existentially quantified formulas of the form $\exists Y. \phi$ where $\phi$ is a gap-order formula over $X$, and $Y \subseteq X$. For $Val : X \rightarrow \mathbb{N}$, we write $Val = \exists Y. \phi$ to denote that there is a mapping $Val' : Y \rightarrow \mathbb{N}$ such that $Val \cup Val' = \phi$. For a (quantified) gap-order formula $\phi$, we define its denotation $\llbracket \phi \rrbracket := \{ Val | Val = \phi \}$.

A gap-order formula $\phi$ over $X$ is in normal form if it satisfies the following conditions:

1. If $(x <_{c_1} y) \in \phi$ and $(y <_{c_2} z) \in \phi$ then $(x <_{c_1 + c_2} z) \in \phi$ for some $c_1$ with $c_1 + c_2 < c_3$.
2. If $(x <_{c} y) \in \phi$ and $(y = z) \in \phi$ (or $(z = y) \in \phi$) then $(x <_{c} z) \in \phi$. 


3. If \((x < y) \in \phi\) and \((x = z) \in \phi\) (or \((z = x) \in \phi\)) then \((z < y) \in \phi\).

4. If \((x = y) \in \phi\) and \((y = z) \in \phi\) then \((z = x) \in \phi\) (or \((z = x) \in \phi\)).

5. For each \(x, y \in X\), there is at most one conjunct in \(\phi\) containing both \(x\) and \(y\).

- **Lemma 1** ([4, 1]). For each gap-order formula \(\phi\), we can effectively compute a gap-order formula \(\phi'\) such that \(\phi'\) is in normal form and \([\phi'] = [\phi]\).

We obtain \(\phi'\) from \(\phi\) by repeatedly adding conjuncts to maintain properties 1-4 and removing conjuncts which violate property 5 (for instance, if we have both \((x < c, y) \in \phi\) and \((x < c, y) \in \phi\), with \(c_1 \leq c_2\), then we can remove the former conjunct.) Normalization can be used to check consistency: the formula is consistent iff no inequalities of the form \(x < x\) are generated.

Furthermore, we can use normalization to perform quantifier elimination as follows. For sets of variables \(Y \subseteq X\) and a gap-order formula \(\phi\), let \(\phi \oplus Y\) to be the gap-order formula we get from \(\phi\) by eliminating all conjuncts in which a variable \(x \in Y\) occurs.

- **Lemma 2** ([4, 1]). Suppose that \(\phi\) is consistent and in normal form. Assume that \(\text{Val} = \phi \oplus Y\). Then there is a \(\text{Val}' : Y \to N\) such that \(\text{Val} \cup \text{Val}' = \phi\).

From Lemma 2 we get the following corollary.

- **Corollary 3.** Suppose that \(\phi\) is consistent and in normal form. Then \([\exists Y. \phi] = [\phi \oplus Y]\).

We write \(\phi_1 \oplus \phi_2\) to denote that \([\phi_2] \subseteq [\phi_1]\). We can check \(\phi_1 \oplus \phi_2\) as follows. By Lemma 1 we can assume that \(\phi_1\) and \(\phi_2\) are in normal form. Then the following conditions should be satisfied: (1) If \((x < c, y) \in \phi_1\) then \((x < c, y) \in \phi_2\) for some \(c_2 \geq c_1\), and (2) if \((x = y) \in \phi_1\) then \((x = y) \in \phi_2\) or \((y = x) \in \phi_2\).

## 5 Traces

We introduce a data structure, called *traces*, that encode *partial runs*. Roughly speaking, a partial run is a run with a number of “holes” inserted. Thus, a partial run consists of a sequence of *segments* each of which is a sequence of consecutive transitions. The source of one transition in a segment is identical to the target of the preceding transition. Furthermore, each push operation in the partial run is matched by a pop operation, and vice versa, such that the push/pop operations respect the stack semantics (i.e., we do not allow push/pop transitions that are “pending” in a partial run). First, we define the set of atomic traces, and describe two operations that allow to build new traces from existing ones. Then, we define an entailment relation on traces. Finally, we use traces to define the notion of *split width*. In the rest of the section, we fix a DMPDA \(\mathcal{A} = (Q, \Delta)\).

### 5.1 Definition

A trace \(\tau\) is a tuple \(\langle \mathbb{I}, \leq, \text{src}, \text{tgt}, E, \phi \rangle\) defined as follows.

- \(\mathbb{I}\) is a finite (index) set. Each index will be used to represent the summary of a segment. The summary is given by the starting and the end states of each segment, i.e., the source of the first transition and the target of the last transition in the segment, and by a relation on the values of the variables before and after performing the different segments. For each variable \(x \in X\) and index \(i \in \mathbb{I}\), we will introduce two new variables \(x_i^1\) and \(x_i^2\) representing the *source* and *target* values of \(x\), i.e., the value of \(x\) at the start and at the end of the segment represented by the index \(i\). We define the set \(X^i := \{x_i^1 (x \in X) \land (i \in \mathbb{I})\} \cup \{x_i^2 (x \in X) \land (i \in \mathbb{I})\}\), and define \(X := \cup_{i \in \mathbb{I}} X^i\).
Atomic Traces

Atomic traces are built using the set of transitions. We will define two types of atomic traces, namely those induced by single internal transitions, and those that are induced by pairs of matching push/pop transitions.

Internal Transitions

Let \( t = (q_1, op, q_2) \in \Delta^\text{intern} \) be an internal transition. We will build a trace with a single index corresponding to a single segment which contains only one transition, namely \( t \). Formally, we define \( \text{MkTrace}(t) := (\mathbb{I}, \preceq, \text{src}, \text{tgt}, E, \phi) \) where

- \( \mathbb{I} = \{ i \} \), i.e., the set of indices is a singleton.
- \( \preceq \) is trivial since the index set contains only one node.
- \( \text{src}(i) = q_1 \) and \( \text{tgt}(i) = q_2 \), i.e., we label the single index with the source and target states of \( t \).
- \( E = \emptyset \) reflecting the fact that the operation performed by \( t \) does not affect the stacks.
- \( \phi \) consists of all conjuncts of the following forms:
  - if \( op = \text{nop} \) or \( op = (x = y) \) or \( op = (x < c y) \) then \( x^A_i = x^4_i \) for all \( x \in \mathbb{X} \), i.e., the values of the variables are not changed during \( t \).
  - if \( op = (x = y) \) then \( x^4_i = y^A_i \), and if \( op = (x < c y) \) then \( y^A_i > x^4_i + c \). The values of the variables should satisfy the condition of the transition.
  - If \( op = (x \leftarrow x) \) or \( op = (x \leftarrow y) \) or \( op = (x \leftarrow (> c y)) \) then \( x^4_z = z^4_i \) for all \( z \in \mathbb{X} - \{ x \} \), i.e., the values of the variables different from \( x \) are not changed during \( t \).
  - If \( op = (x \leftarrow y) \) then \( x^4_i = y^A_i \).
  - If \( op = (x \leftarrow (> c y)) \) then \( x^4_i > y^A_i + c \).
Stack Transitions

Consider two transitions \( t_1 = (q_1, op_1, q_2) \) \( \in \Delta_{\text{push}}^\sigma \), \( t_2 = (q_3, op_2, q_4) \) \( \in \Delta_{\text{pop}}^\sigma \) where \( op_1 = \text{push}(\sigma)(a)(x) \) and \( op_2 = \text{pop}(y)(\sigma)(a) \). Notice that the two transitions push/pop the same symbol \( a \) to/from the same stack \( \sigma \). We will build a trace with two indices corresponding to two segments containing \( t_1 \) and \( t_2 \), respectively. Formally, we define \( \text{MkTrace}(t_1, t_2) := (I, s, \text{src}, \text{tgt}, E, \phi) \), where:

- \( I = \{i_1, i_2\} \) i.e., the index set contains two elements. We use the indices \( i_1 \) and \( i_2 \) to represent two segments each containing a single transition, namely \( t_1 \) and \( t_2 \) respectively.
- \( i_1 \leq i_2 \). We require that \( i_1 \) is ordered before \( i_2 \). This reflects the fact that a pop transition occurs after the matching push transition.
- \( E(i_1, i_2) = \{\sigma\} \), i.e., we add an edge between \( i_1 \) to \( i_2 \) labeled with \( \sigma \) corresponding to the matching push/pop operations on \( \sigma \) performed by \( t_1 \) resp. \( t_2 \).
- \( \text{src}(i_1) = q_1 \), \( \text{tgt}(i_1) = q_2 \), \( \text{src}(i_2) = q_3 \), and \( \text{tgt}(i_2) = q_4 \). In other words, we label the new indices with the source and target states of \( t_1 \) resp. \( t_2 \).
- \( \phi \) consists of all conjuncts of the following forms:
  (i) \( x_z^1 = x_z^2 \) for all \( z \in X \), i.e., the values of the variables are not changed during \( t_1 \).
  (ii) \( z_x^2 = z_x^2 \) for all \( z \in X - \{y\} \), i.e., the values of the variables, except \( y \), are not changed during \( t_2 \).
  (iii) \( y_z^2 = y_z^2 \).

This condition corresponds to the fact that the value of \( a \) when pushed to the stack during \( t_1 \) is equal to the value of variable \( x \). This value is identical to the value stored in \( y \) after the pop operation of transition \( t_2 \).

5.3 Operations

We define two operations for building new traces, namely shuffling and contraction.

Shuffling

Consider two traces \( \tau_1 = \langle I_1, \leq_1, \text{src}_1, \text{tgt}_1, E_1, \phi_1 \rangle \), and \( \tau_2 = \langle I_2, \leq_2, \text{src}_2, \text{tgt}_2, E_2, \phi_2 \rangle \), where \( I_1 \cap I_2 = \emptyset \). We will build a new trace by shuffling the index sets of \( \tau_1 \) and \( \tau_2 \). We define \( \tau_1 \otimes \tau_2 \) to be the set of traces of the form \( (I, s, \text{src}, \text{tgt}, E, \phi) \) satisfying the following conditions:

- \( I = I_1 \cup I_2 \), i.e., the new trace contains exactly all the segments that are in \( \tau_1 \) and \( \tau_2 \).
- \( \leq \) is a total ordering on \( I \) such that \( \leq_1 \leq \) and \( \leq_2 \leq \). We do not change the original orderings of the indices, but we do not constrain the places of the two sets of indices relative to each other.

- \( \text{src}(i) = \text{src}_1(i) \) and \( \text{tgt}(i) = \text{tgt}_1(i) \) for all \( i \in I_1 \). Furthermore, \( \text{src}(i) = \text{src}_2(i) \), and \( \text{tgt}(i) = \text{tgt}_2(i) \) for all \( i \in I_2 \). (We keep the state labelings of the old indices.)

- \( E(i_1, i_2) = E_1(i_1, i_2) \) if \( i_1, i_2 \in I_1 \), and \( E(i_1, i_2) = E_2(i_1, i_2) \) if \( i_1, i_2 \in I_2 \), i.e., all the edges in \( \tau_1 \) and \( \tau_2 \) are maintained in \( \tau_1 \otimes \tau_2 \). Also, \( E(i_1, i_2) = \emptyset \), if \( i_1 \in I_1 \) and \( i_2 \in I_2 \) or if \( i_1 \in I_2 \) and \( i_2 \in I_1 \), i.e., we do not add any edges between the two sets of indices. Furthermore, we require that there are no \( i_1, i_2 \in I_1 \) and \( i_3, i_4 \in I_2 \) such that
  (i) \( \sigma \in E_1(i_1, i_2) \),
  (ii) \( \sigma \in E_2(i_3, i_4) \), and
  (iii) either \( i_1 < i_3 < i_2 < i_4 \) or \( i_3 < i_1 < i_4 < i_2 \).

This is to ensure that \( \tau_1 \otimes \tau_2 \) respects the stack semantics.

- \( \phi = \phi_1 \wedge \phi_2 \). Notice that the values of the variables indexed by elements from \( I_1 \) are not related to the values of the variables indexed by elements from \( I_2 \).
Contraction

We define a contraction operation $\downarrow$ on indices that represents merging the corresponding segments. Consider $i_1, i_2 \in I$ such that $i_1 \prec i_2$, i.e., $i_2$ is the immediate successor of $i_1$. Let $\text{src}(i_1) = q_1$, $\text{tgt}(i_1) = q_2$, $\text{src}(i_2) = q_2$, and $\text{tgt}(i_1) = q_3$, i.e., the target state of $i_1$ is identical to the source state of $i_2$. We will merge $i_1$ and $i_2$ to a new (single) index $j$. We define $\tau \downarrow (i_1, i_2) := (\tau', s', \text{src}', \text{tgt}', E', \phi')$ as follows:

1. $\tau' = \tau - i_1, i_2 \cup \{j\}$ where $j \notin I$, i.e., we replace the two merged indices by a new one.
2. $k \preceq j$ iff $k \preceq i_1$, and $j \preceq k$ iff $i_2 \preceq k$ for all $k \in I - \{i_1, i_2\}$, i.e., in the new ordering, the new index $j$ will take the places of the two (consecutive) indices $i_1$ and $i_2$. Furthermore, $k_1 \preceq j$ iff $k_1 \preceq k_2$ for all $k_1, k_2 \in I - \{i_1, i_2\}$, i.e., the relative orderings of the original indices are not changed.
3. $\text{src}'(j) = q_1$, $\text{tgt}'(j) = q_3$, $\text{src}'(k) = \text{src}(k)$ and $\text{tgt}'(k) = \text{tgt}(k)$ for all $k \notin I - \{i_1, i_2\}$.
4. $E'(j, k) = E(i_1, k) \cup E(i_2, k)$ and $E'(k, j) = E(k, i_1) \cup E(k, i_2)$ for all $k \notin I - \{i_1, i_2\}$, i.e., we merge the edges originating from/to the two merged indices. Also, $E'(k_1, k_2) = E(k_1, k_2)$ for all $k_1, k_2 \in I - \{i_1, i_2\}$, i.e., the edges to/from the other indices are maintained. Notice that any edges between $i_1$ and $i_2$ are deleted.

- $\phi' = \exists \left( x_1^1 \cup x_2^1 \right). \phi''$, where $\phi''$ is defined as $\phi \land (\land_{x_1^1} (x_1^1 = x_2^1)) \land (\land_{x_2^1} (x_1^1 = x_2^1))$. We require that $\phi'$ is consistent. Since we merge the two segments corresponding to $i_1$ and $i_2$, we require the values of the variables at the end of the first segment to be consistent with the values of the variables at the start of the second segment. If these conditions are not satisfied, then the resulting formula $\phi'$ will not be consistent. Furthermore, the values of the variables at the start of the new segment are defined to be the values of the variables at the start of the segment corresponding to $i_1$. Analogously, the values of the variables at the end of the new segment are defined to be the values of the variables at the end of the segment corresponding to $i_2$. By Lemma 1, we know that $\phi''$ can be transformed to an equivalent formula in normal form, and hence by Corollary 3, we can compute $\phi'$ as a gap-order formula.

We define $\tau \downarrow$ to be the set of traces $\tau'$ such that: (1) there are indices $i_1, i_2 \in I$ with $i_1 \prec i_2$, (2) $\text{tgt}(i_1) = \text{src}(i_2)$, and (3) $\tau' = \tau \downarrow (i_1, i_2)$.

5.4 Entailment

We define an entailment relation $\models$ on traces. Intuitively, a trace $\tau_1$ is weaker than a trace $\tau_2$, denoted $\tau_1 \preceq \tau_2$, if their graphs are isomorphic (equivalent up to the renaming of indices), but the gap-order formula of $\tau_1$ is weaker than the one of $\tau_2$. Later in the paper, when we compute reachability relations, we let $\tau_1$ “subsume” $\tau_2$ in the sense that if we encounter both $\tau_1$ and $\tau_2$ then we only include $\tau_1$ (and discard $\tau_2$), without suffering any loss of any precision in the analysis. Formally, consider traces $\tau_1 = (I_1, s_1, \text{src}_1, \text{tgt}_1, E_1, \phi_1)$ and $\tau_2 = (I_2, s_2, \text{src}_2, \text{tgt}_2, E_2, \phi_2)$. Let $h : I_1 \to I_2$ be a bijection. We write $\tau_1 \preceq_h \tau_2$ to denote that the following conditions are satisfied: (1) $i_1 \preceq h(i_1)$ for all $i_1 \in I_1$, (2) $\text{src}_1(i) = \text{src}_2(h(i))$ and $\text{tgt}_1(i) = \text{tgt}_2(h(i))$, (3) $E_1(i_1, i_2) = E_2(h(i_1), h(i_2))$, and (4) $\phi_1^h \equiv \phi_2$. We obtain $\phi_1^h$ from $\phi_1$ by replacing each $x_k^1$ by $x_k^{h(1)}$ and replacing each $x_k^2$ by $x_k^{h(2)}$. We use $\tau_1 \preceq \tau_2$ to denote that $\tau_1 \preceq_h \tau_2$ for some $h$. 
5.5 Split Width

We say that a trace has split width $\theta$ if it can be derived by starting from the atomic traces and repeatedly applying the shuffling and contraction operations without letting any of the intermediate-generated traces have a degree larger than $\theta$.

A proof tree $T$ of split width $\theta$ is a binary tree whose nodes are labeled with traces such that the following conditions are satisfied:

- The leaves are labeled with atomic traces.
- If an internal node, labeled with $\tau$, has two children then the children are labeled with traces $\tau_1$ and $\tau_2$ such that $\tau \in \tau_1 \otimes \tau_2$.
- If an internal node, labeled with $\tau'$, has a single child then that child is labeled with a trace $\tau$ such that $\tau \in \tau' \downarrow$.
- For each label $\tau$ of node in $T$, we have $\#\tau \leq \theta$.

A trace $\tau$ is of split width $\theta$ if $\theta$ is the smallest number such that $\tau$ is the root of a proof tree of split width $\theta$. We use $SW(\tau)$ to denote the split width of $\tau$.

We extend the notion of split width to plain runs as follows. Consider a plain run $\pi = c_0 \ell_1 c_1 \ldots c_{n-1} \ell_n c_n$ where $c_i = (q_i, \alpha_i, \beta_i)$. We define $\bar{\pi}$ to be the set of traces of degree one, of the form $\{ (1) \cdot \ell \cdot \mathsf{src} \cdot (t, E, \phi) \}$ where $s \subseteq \{ (1,1) \}$, $\mathsf{src} (1) = q_0$, $\mathsf{tgt} (1) = q_n$, $E = \emptyset$, and $\mathsf{Val} = \phi$ with $\mathsf{Val} (x^i_0) = \alpha_0 (x)$ and $\mathsf{Val} (x^i_n) = \alpha_n (x)$ for all $x \in X$. We define $\mathsf{SW}(\pi)$ to be the smallest $k \in \mathbb{N}$ such that there is a trace $\tau \in \bar{\pi}$ with $SW(\tau) = k$. For states $q, q' \in Q$ and $\theta \in \mathbb{N}$, we define $\mathcal{R}^\theta(q, q') := \{ \langle \alpha, \alpha' \rangle \mid 3\pi. (\mathsf{SW}(\pi) \leq \theta) \land (q, \alpha, \beta) \xrightarrow{\tau} (q', \alpha', \beta') \}$.

For a DMPDA $A$, we define the split width $\mathsf{SW}(A)$ to be the largest $k$ such that there is a plain run $\pi$ in $A$ with $\mathsf{SW}(\pi) = k$. For a class $C$ of DMPDA, we define $\mathsf{SW}(C)$ to be the largest $k$ such that there is an $A \in C$ with $\mathsf{SW}(A) = k$. We say that $C$ has bounded split width if $\mathsf{SW}(C) = k$ for some $k \in \mathbb{N}$.

6 Decidability

In this section, we present the main result of the paper:

\begin{itemize}
  \item \textbf{Theorem 4.} The reachability problem is decidable for any class of DMPDA with bounded split width.
\end{itemize}

To prove Theorem 4 we first describe an algorithm computing reachability relations, and then prove its correctness in Lemma 5 (termination), Lemma 6 (soundness), and Lemma 7 (completeness).

6.1 Algorithm

The algorithm inputs a DMPDA $A = (Q, \Delta)$ together with an upper limit $\theta$ on the degrees of the traces to be considered during the analysis. The algorithm maintains a set $W$ of traces that have been detected but not analyzed, and a set $V$ of traces that have been both detected and analyzed. Initially, the sets $W$ and $V$ are empty (Line 1–2). We add all the atomic traces induced by internal transitions (Line 3), and by matching push/pop transitions (Line 5) to the set $W$. After the initialization phase, the algorithm performs a number of iterations using the repeat-loop of Line 8. In each iteration, we first select and remove an element $\tau$ from $W$ (Lines 9–10). We check that $\tau$ satisfies two conditions (Line 11), namely: that (i) the degree of $\tau$ is within the allowed limit, and that (ii) $\tau$ is not subsumed by any trace already in the set $V$. If the two conditions are satisfied, we use $\tau$ to generate new traces to analyze. These new traces are added to the set $W$. First, we take the shuffle of $\tau$ with each member of the set
Algorithm 1: Computing the Reachability Relation.

**Input:** \( A = (Q, \Delta) \): DMPDA,
\( \theta \): maximal index size

**Output:** characterization of the reachability relation

1. \( V \leftarrow \emptyset \);
2. \( W \leftarrow \emptyset \);
3. for each \( t \in \Delta_{\text{intern}} \) do
4. \( W \leftarrow W \cup \{ \text{MkTrace}(t) \} \)
5. for each \( t_1 \in \Delta_{\text{push}} \) and \( t_2 \in \Delta_{\text{pop}} \) do
6. if \( \sigma_1 = \sigma_2 \) and \( a_1 = a_2 \) then
7. \( W \leftarrow W \cup \{ \text{MkTrace}(t_1, t_2) \} \)
8. repeat
9. select some \( \tau \in W \);
10. \( W \leftarrow W - \{ \tau \} \);
11. if \( (\#\tau \leq \theta) \land (\exists \tau' \in W. \tau' \subseteq \tau) \) then
12. for each \( \tau' \in V \) do
13. \( W \leftarrow W \cup (\tau \otimes \tau') \)
14. \( W \leftarrow W \cup \tau \)
15. \( V \leftarrow (\tau' \in V \land \tau \neq \tau') \cup \{ \tau \} \);
16. until \( W = \emptyset \);
17. for each \( q, q' \in Q \) do
18. \( \mathcal{R}(q, q') \leftarrow \emptyset \);
19. for each \( \tau \in V \) do
20. let \( \tau = (I, \leq, \text{src}, \text{tgt}, E, \phi) \);
21. if \( \#\tau = 1 \) then
22. \( \mathcal{R}(q, q') \leftarrow \mathcal{R}(q, q') \cup \{ \phi \} \)
24. return \( \mathcal{R} \)

V (Line 12). Then, we add all possible contractions of \( \tau \) (Line 14). Finally, at Line 15, we add \( \tau \) to \( V \), and at the same time remove all elements of \( V \) that are subsumed by \( \tau \). Notice that this means that all the traces in the set \( V \) will be pairwise incomparable wrt. \( \preceq \). The iteration of the main loop is repeated until the set \( W \) becomes empty.

After the termination of the loop, we build the reachability relations successively by going through all traces that have been added to \( V \) (Lines 17–23). For each pair of states \( q \) and \( q' \), we maintain a set \( \mathcal{R}(q, q') \) of gap-order formulas. Each time a trace \( \tau \) of degree one is encountered in \( V \), we add the gap-order formula of \( \tau \) to the set \( \mathcal{R}(q, q') \) corresponding the source state \( q \) and target state \( q' \) of the (only) index of \( \tau \). At the end of the algorithm, the reachability relation between any pair of states is characterized by the union of the denotations of all the gap-order formulas in the corresponding set.

### 6.2 Correctness

We show correctness of the algorithm in several steps. We start by showing that the algorithm always terminates. For a set \( A \), a pre-order \( \preceq \) on \( A \) is said to be a **Well Quasi-Ordering**
(WQO) if, for each infinite sequence $a_0 a_1 a_2 \cdots$ of elements from $A$, there are $i < j$ such that $a_i \neq a_j$. For a set $T$ of traces, we define its degree $\#T := \max_{\tau \in T} (\#\tau)$. Notice that $\#T$ need not exist in general. Gap-order formulas over $X$ is WQO under the $\subseteq$ relation [1, 4]. Hence, for any $k \in \mathbb{N}$ and any set of traces $T$ with $\#T \leq k$, the entailment relation $\models$ is a WQO on $T$. This gives the following lemma.

- **Lemma 5.** The algorithm is guaranteed to terminate.

Next, we show soundness and completeness of the algorithm. To that end, we will consider the set $\mathcal{R}(q, q')$ of gap-order formulas for each pair of states $q$ and $q'$, returned by the algorithm. We define a denotation function for these formulas, and relate them to the reachability relation $\mathbb{R}(q, q')$. We show that each member in the denotation corresponds to a run (Lemma 6) implying the soundness of the algorithm. Conversely, we show that each run with split width $\theta$ belongs to the denotation (Lemma 7) implying the completeness of the algorithm.

A formula $\phi$ is said be transitional over $X$ if $\phi$ is a gap-order formula over $X^1$ where $\parallel \parallel = 1$. Notice that, if $I = \{i\}$, then each variable in $\phi$ is either of the form $x_i^0$ or of the form $x_i^1$ where $x \in X$. We define $\parallel \parallel$ to be the set of pairs $(\alpha, \alpha')$ such that $\alpha : X \rightarrow \mathbb{N}, \alpha' : X \rightarrow \mathbb{N}$, and there is a $\mathsf{Val} : X \rightarrow \mathbb{N}$ with $\mathsf{Val} = \phi$, $\alpha(x) = \mathsf{Val}(x_i^0)$ and $\alpha'(x) = \mathsf{Val}(x_i^1)$ for all $x \in X$. For a set of $\Phi$ of transitional gap-order formulas, we define $\parallel \parallel \Phi := \bigcup_{\phi \in \Phi} \parallel \parallel \phi$. Notice that all members of $\mathcal{R}(q, q')$ are transitional gap-order formulas. The following two lemmas then show the soundness and completeness of the algorithm.

- **Lemma 6.** $\forall q, q' \in Q. \parallel \mathcal{R}(q, q') \parallel \subseteq \mathcal{R}(q, q')$
- **Lemma 7.** $\forall q, q' \in Q. \parallel \mathcal{R}(q, q') \parallel \supseteq \mathbb{R}^\Phi(q, q')$

### 7 Applications

#### 7.1 Pushdown automata

A data push-down automata is a DMPDA with a single stack. All plain runs of a data-push-down automaton have split-width bounded by 3 [21]. Thus, it follows from Theorem 4 that

- **Corollary 8.** The reachability problem is decidable for data push-down automata.

#### 7.2 Multi-push-down systems

As already mentioned in the introduction, the control state reachability is undecidable even in a finite data setting for multi-pushdown systems. Several under-approximation classes (cf. Introduction) have been proposed in the literature for regaining decidability in the finite data case. We recall their definitions below.

- **Bounded context-switch** [30] A context is a sequence of operations in which at most one stack is touched. A run is $k$-context bounded if it is the concatenation of at most $k$ contexts.

- **Bounded phase** [25] A phase is a sequence of operations in which at most one stack can be popped, though there are no restrictions on pushes. An run is $k$-phase bounded if it is the concatenation of at most $k$ phases. This subsumes the $k$-context bounded runs.

- **Ordered stacks** [10, 9]. In an ordered multi-pushdown system, the stacks are ordered linearly by priority. Further a stack may be popped only if all the stacks with higher
priority are empty. An ordered-stack run respects the priority ordering in its stack operations.

- **Bounded scope** [26] A run of a multi-pushdown system is \( k \)-scope bounded if the number of context switches between a push and the corresponding pop is always bounded by \( k \). This definition is slightly general than the original round-based definition of [26]. This subsumes the \( k \)-context bounded runs, but is orthogonal to \( k \)-phase bounded runs.

Runs falling into any of the above class have bounded split-width.

- **Fact 1 ([21]).** Split-width of \( k \)-context bounded runs is at most \( k + 2 \).
- Split-width of \( k \)-phase bounded runs is at most \( 2^k \).
- Split-width of \( k \)-scope bounded runs is at most \( k + 2 \).
- Split-width of ordered-stack runs is at most \( 2^{[3]} \).

Let \( U \) be an under-approximation class above. A \( U \)-run of a DMPDA \( A \) is a run that is in \( U \). For the under-approximation \( U \), let \( C_U \) be the class of DMPDA such that all runs of any DMPDA \( A \in C_U \) are \( U \)-runs. By Fact 1, \( C_U \) has bounded split-width. Hence by Theorem 4, we get

- **Corollary 9.** The reachability problem is decidable for \{ bounded context-switching, bounded phase, ordered stacks, bounded scope \} - DMPDA.

Let \( U \) be an under-approximation class above. The \( U \)-reachability problem asks, given a DMPDA \( A \) and two plain configurations \( c_1 \) and \( c_2 \), whether it is possible to reach \( c_2 \) from \( c_1 \) by a \( U \)-run.

If the input DMPDA \( A \) belongs to \( C_U \), then \( U \)-reachability problem is the same as the reachability problem, which is decidable by Corollary 9. However, an arbitrary \( A \) may have runs outside of \( U \) in general, but still having bounded split-width. Thus running our algorithm naively on any input \( A \) could say “yes” to a pair of configurations \( c_1 \) and \( c_2 \) even when they are not \( U \)-reachable.

In order to decide the \( U \)-reachability problem, given the class \( U \) and DMPDA \( A \), we will construct a new DMPDA \( A' \in C_U \). The DMPDA \( A' \) in effect will enforce the semantic restriction of \( U \)-runs syntactically into the automaton \( A \). Thus runs of \( A' \) are precisely \( U \)-runs of \( A \). We will thus reduce the \( U \)-reachability problem in \( A \) to the reachability problem in \( A' \in C_U \) which is decidable by Corollary 9.

More formally, we reduce the \( U \)-reachability problem to reachability problem in \( C_U \). The reduction depends on the under-approximation \( U \). On input DMPDA \( A \), plain configurations \( c_1 \) and \( c_2 \), we will construct a DMPDA \( A' \) belonging to \( C_U \). Then we will compute a finite set of pairs of plain configurations \( c'_1 \) and \( c'_2 \) from \( c_1 \) and \( c_2 \). We check whether \( c'_2 \) is reachable from \( c'_1 \) in \( A' \) for at least one pair of computed plain configurations \( c'_1 \) and \( c'_2 \). If this is the case, we conclude that \( c_2 \) is \( U \)-reachable from \( c_1 \) in \( A \). Otherwise, \( c_2 \) is not \( U \)-reachable from \( c_1 \) in \( A \).

Due to lack of space, we will now describe the reduction in detail for only the case of bounded-phase.

**Reduction for \( k \)-phase bounded.**

Given a DMPDA \( A = (Q, \Delta) \) and a bound \( k \) on the number phases, we construct a new one \( A' = (Q', \Delta') \) where \( Q' = Q \times \{1 \ldots k\} \times \Sigma \cup Q \times \{0\} \). The state remembers, in addition to the state of \( A \), how many phases have been used so far, and the stack that is being popped from in the current phase. Thus a state of the form \((q, i, \sigma)\) means that currently \( A \) would
have been in state \( q \), and in the \( i \)th phase, which is allowed to pop only from stack \( \sigma \). The state \((q, 0)\) is used to start off a computation, where no stack has been popped yet. The transitions \( \Delta' \) lifts \( \Delta \) to smoothly extend to \( Q' \) while maintaining the intended semantics. For instance, if \((q_1, \text{op}, q_2) \in \Delta \), then \( (q_1, i, \sigma), \text{op}, (q_2, i, \sigma) \in \Delta' \) and \((q_1, 0), \text{op}, (q_2, 0) \in \Delta' \) if \( \text{op} \) is an operation of the forms (i) to (vii) (cf. Section 3). If \((q_1, \text{op}, q_2) \in \Delta \) and if \( \text{op} \) is of the form (viii), i.e., \( \text{pop}(x)(\sigma)(a) \), then we have i) \((q_1, i, \sigma), \text{op}, (q_2, i, \sigma) \in \Delta' \), ii) \((q_1, i, \sigma'), \text{op}, (q_2, i + 1, \sigma) \in \Delta' \) if \( \sigma' = \sigma \) and \( i < k \), and, iii) \((q_1, 0), \text{op}, (q_2, 1, \sigma) \in \Delta' \).

The DMPDA \( \mathcal{A}' \) exhibits all and only executions of \( \mathcal{A} \) in which the number of phases is bounded by \( k \). Given the pair of plain configurations \( c_1 = (q_1, \alpha_1, \beta_1) \) and \( c_2 = (q_2, \alpha_2, \beta_2) \) of \( \mathcal{A} \), we obtain the set of pairs of the form \((q_1', \alpha_1, \beta_1), (q_2', \alpha_2, \beta_2)\) where \( q_1' = (q_1, 0) \) and \( q_2' \) is either \((q_2, 0)\) or of the form \( q_2' = (q_2, i, \sigma) \) for some \( i \) and \( \sigma \).

If \( c_2' = (q_2', \alpha_2, \beta_2) \) is reachable from \( c_1' = (q_1', \alpha_1, \beta_1) \) in \( \mathcal{A}' \) for one such computed pair, then indeed, \( c_2 \) is \( k \)-phase reachable from \( c_1 \) in \( \mathcal{A} \). Conversely, if \( c_2 \) is \( k \)-phase reachable from \( c_1 \) in \( \mathcal{A} \), then there exists \( c_2' \) and \( c_1' \) of the form described above such that \( c_2' \) is reachable from \( c_1' \) in \( \mathcal{A}' \). This concludes our reduction.

\[ \text{Corollary 10.} \quad \text{The \( k \)-phase reachability problem is decidable for any DMPDA.} \]

In a similar manner, we can have reductions for each of the under-approximations described above. Thus, we get

\[ \text{Corollary 11.} \quad \{\text{Bounded context-switching, bounded phase, ordered stacks, bounded scope}\}-\text{reachability problem is decidable for any DMPDA.} \]

## 8 Conclusions

We have studied the reachability problem for multi-pushdown systems with gap-order constraints. We provide an algorithm for solving the reachability problem. The algorithm is sound and complete for the classes of automata that have a bounded split-width.

For future work, we plan to consider lifting the framework to a more general setting of auxiliary storages which include queues and multi-sets. Furthermore, it would be interesting to consider the case of distributed processes.

### References

Data Multi-Pushdown Automata


Towards an Efficient Tree Automata Based Technique for Timed Systems

S. Akshay¹, Paul Gastin², Shankara Narayanan Krishna³, and Ilias Sarkar⁴

¹ Dept of CSE, IIT Bombay, India
akshayss@cse.iitb.ac.in

² LSV, ENS Paris-Saclay, CNRS, France
paul.gastin@lsv.fr

³ Dept of CSE, IIT Bombay, India
krishnas@cse.iitb.ac.in

⁴ Dept of CSE, IIT Bombay, India
ilias@cse.iitb.ac.in

Abstract

The focus of this paper is the analysis of real-time systems with recursion, through the development of good theoretical techniques which are implementable. Time is modeled using clock variables, and recursion using stacks. Our technique consists of modeling the behaviours of the timed system as graphs, and interpreting these graphs on tree terms by showing a bound on their tree-width. We then build a tree automaton that accepts exactly those tree terms that describe realizable runs of the timed system. The emptiness of the timed system thus boils down to emptiness of a finite tree automaton that accepts these tree terms. This approach helps us in obtaining an optimal complexity, not just in theory (as done in earlier work e.g. [4]), but also in going towards an efficient implementation of our technique. To do this, we make several improvements in the theory and exploit these to build a first prototype tool that can analyze timed systems with recursion.

1998 ACM Subject Classification F.1.1 Models of Computation

Keywords and phrases Timed automata, tree automata, pushdown systems, tree-width

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2017.39

1 Introduction

Development of efficient techniques for the verification of real time systems is a practically relevant problem. Timed automata [6] are a prominent and well accepted abstraction of timed systems. The development of this model originally began with highly theoretical results, starting from the PSPACE-decision procedure for the emptiness of timed automata. But later, this theory has led to the development of state of the art and industrial strength tools like UPPAAL [7]. Currently, such tools are being adapted to build prototypes that handle other systems such timed games, stochastic timed systems etc. While this helps in analysis of certain systems, there are complicated real life examples that require paradigms like recursion, multi-threaded concurrency and so on.

* This work was partly supported by UMI-ReLaX, DST-CEFIPRA project AVeRTS and DST-INSPIRE faculty award [IFA12-MA-17].
Towards an Efficient Tree Automata Based Technique for Timed Systems

For timed systems with recursion, a popular theoretical framework is the model of timed pushdown automata (TPDA). In this model, in addition to clock variables as in timed automata, a stack is used to model recursion. Depending on how clocks and stack operations are integrated, several variants [8], [1], [14], [12], [9] have been looked at. For many of these variants, the basic problem of checking emptiness has been shown decidable (and \textsc{EXPTIME}-complete) using different techniques. The proofs in [8], [1], [14] work by adapting the technique of region abstraction to untime the stack and obtain a usual untimed pushdown automaton, while [9] gives a proof by reasoning with sets of timed atoms. Recently, in [4], a new proof technique was introduced which modeled the behaviours of the TPDA as graphs with timing constraints and analyzed these infinite collections of time-constrained graphs using tree automata. This approach follows the template which has been explored in depth for various untimed systems in [13], [11], [3]. The basic idea can be outlined as follows: (1) describe behaviours of the underlying system as graphs, (2) show that this class of graphs has bounded width, (3) either appeal to Courcelle’s theorem [10] by showing that the desired properties are MSO-defineable or explicitly construct a tree-automaton to capture the class of graphs that are the desired behaviours. The work in [4] extends this approach to timed systems, by considering their behaviors as time-constrained words. The main difficulty here is to obtain a tree automaton that accepts only those time-constrained words that are realizable via a valid time-stamping.

Despite the amount of theoretical work in this area [8, 13, 11, 4, 1, 9], none of these algorithms have been implemented to the best of our knowledge. Applying Courcelle’s theorem is known to involve a blowup in the complexity (depending on the quantifier-alternation of the MSO formula). The algorithm for checking emptiness in [4] for the timed setting which directly constructs the tree automaton avoiding the MSO translation also turns out to be unimplementable even for small examples due to the following reasons:

First, it has a pre-processing step where each transition in the underlying automaton is broken into several micro transitions, one for each constraint that is checked there, and one corresponding to each clock that gets reset on that transition. This results in a blowup in the size of the automaton. Second, the number of states of the tree automaton that is built to check realizability as well as the existence of a run of a system is bounded by $(M \times T)^{O(K^2)} 2^{O(K^2 \lg K)}$, where $M$ is one more than the maximal constant used in the given system, $T$ is the number of transitions, and $K = 4|X| + 6$ is the so-called split-width, where $|X|$ is the number of clocks used. This implies that even for a system that has 1 clock, 5 transitions and uses a maximum constant 5, we have more than $30^{100}$ states.

In this paper, we take the first steps towards an efficient implementation. While we broadly follow the graph and tree-automata based approach (and in particular [4]), our main contribution is to give an efficient technique for analyzing TPDA. This requires several fundamental advances: (i) we avoid the preprocessing step, obtaining a direct bound on tree width for timed automata and TPDA. This is established by playing a split-game which decomposes the graph representing behaviours of the timed system into tree terms; by coloring some vertices of the graph and removing certain edges whose endpoints are colored. The minimum number of colors used in a winning strategy is 1 plus the tree-width of the graph. (ii) we develop a new algorithm for building the tree automaton for emptiness, whose complexity is in \textsc{ETIME}, i.e., bounded by $(M \times T)^{3|X| + 3}$ with an exponent which is a linear function of the input size (improved from \textsc{EXPTIME}, where the exponent is a polynomial function of the input). Thus, if the system has 1 clock, 5 transitions and uses a maximum constant 5, we have only $\sim 30^6$ states. In particular, our tree-automaton is strategy-driven, i.e., it manipulates only those tree terms that arise out of a winning strategy of our split-game. As a result of this strategy-guided approach, the number of states of our
tree automaton is highly optimized, and an accepting run exactly corresponds to the moves in a winning strategy of our split-game. (iii) Finally, our algorithm outputs a witness for realizability (and non-emptiness). As a proof-of-concept, we implemented our algorithm and despite the worst-case complexity, we discuss optimizations, results and a modeling example where our implementation performs well.

Due to lack of space, we have not included all the proofs here; the full version of the paper, with proofs of all the results, illustrative examples, details of experimental results and benchmarks used can be found at [5].

2 Graphs for behaviors of timed systems

We fix an alphabet $\Sigma$ and use $\Sigma_\varepsilon$ to denote $\Sigma \cup \{\varepsilon\}$, where $\varepsilon$ is the silent action. We also fix a finite set of intervals $I$ with bounds in $\mathbb{N} \cup \{\infty\}$. For a set $S$, we use $\leq S \times S$ to denote a partial or total order on $S$. For any $x, y \in S$, we write $x < y$ if $x \leq y$ and $x \neq y$, and $x < y$ if $x < y$ and there does not exist $z \in S$ such that $x < z < y$.

Definition 1. A word with timing constraints (TCW) over $(\Sigma, I)$ is a structure $V = (V, \rightarrow, \lambda, (\wedge I)_{I \in I})$ where $V$ is a finite set of vertices or positions, $\lambda : V \rightarrow \Sigma_\varepsilon$ labels each position, the reflexive transitive closure $\lessgtr I \rightarrow \lessgtr I$ is a total order on $V$ and $\rightarrow = \lessgtr I$ is the successor relation, while $\wedge I \lessgtr I$ connects pairs of positions carrying a timing constraint, given by an interval in $I \in I$. A TCW $V = (V, \rightarrow, \lambda, (\wedge I)_{I \in I})$ is called realizable if there exists a timestamp map $\text{ts} : V \rightarrow \mathbb{R}_+$ such that $\text{ts}(i) \leq \text{ts}(j)$ for all $i \leq j$ (time is non-decreasing) and $\text{ts}(j) - \text{ts}(i) \in I$ for all $i \lessgtr I j$ (timing constraints are satisfied).

An example of a TCW is given in Figure 1 (right), with positions 0, 1, 2, 3 labelled by $\Sigma = \{a, b, c\}$. Curved edges decorated with intervals connect positions related by $\wedge I$, while straight edges define the successor relation $\rightarrow$. This TCW is realizable by the sequence of timestamps 0, 0.9, 2.89, 3.1 but not by 0, 0.9, 2.99, 3.1. We let $\text{Real}(\Sigma, I)$ be the set of TCWs over $(\Sigma, I)$ which are realizable.

TPDA and their semantics as TCWs. Dense-timed pushdown automata (TPDA), introduced in [1], are an extension of timed automata, and operate on a finite set of real-valued clocks and a stack which holds symbols with their ages. The age of a symbol represents the time elapsed since it was pushed onto the stack. Formally, a TPDA $S$ is a tuple $(S, s_0, \Sigma, \Lambda, \Delta, X, F)$ where $S$ is a finite set of states, $s_0 \in S$ is the initial state, $\Sigma$, $\Lambda$, are respectively finite sets of input, stack symbols, $\Delta$ is a finite set of transitions, $X$ is a finite set of real-valued variables called clocks, $F \subseteq S$ are final states. A transition $t \in \Delta$ is a tuple $(s, \gamma, a, \text{op}, R, s')$ where $s, s' \in S$, $a \in \Sigma$, $\gamma$ is a finite conjunction of atomic formulae of the kind $x \in I$ for $x \in X$ and $I \in I$, $R \subseteq X$ are the clocks reset, $\text{op}$ is one of the following stack operations:

1. $\text{nop}$ does not change the contents of the stack,
2. $\downarrow c, c \in \Lambda$ is a push operation that adds $c$ on top of the stack, with age 0.
3. $\uparrow c, c \in \Lambda$ is a stack symbol and $I \in I$ is an interval, is a pop operation that removes the top most symbol of the stack provided it is a $c$ with age in the interval $I$.

Timed automata (TA) can be seen as TPDA using $\text{nop}$ operations only. This definition of TPDA is equivalent to the one in [1], but allows checking conjunctive constraints and stack operations together. In [9], it is shown that TPDA of [1] are expressively equivalent to timed automata with an untimed stack. As our technique is oblivious to whether the stack is timed or not, we focus on the syntactically more succinct model TPDA with a timed stack.

Next, we define the semantics of a TPDA in terms of TCWs.
Towards an Efficient Tree Automata Based Technique for Timed Systems

non-decreasing sequence of values in a timed word is a sequence

Remark. Figure 1 depicts a simple example of a timed automaton and a

Definition 2. A TCW $\mathcal{V} = (V, \rightarrow, \lambda, (\cap^I)_{I \in \mathcal{I}})$ is said to be generated or accepted by a

We denote by $\mathcal{TCW}(S)$ the set of TCWs generated by $S$. The non-emptiness problem for the

Figure 1 depicts a simple example of a timed automaton and a TCW generated by it.

Remark. The classical semantics of timed systems is given in terms of timed words. A

We now identify some important properties satisfied by TCWs generated from a TPDA.

Let $\mathcal{V} = (V, \rightarrow, \lambda, (\cap^I)_{I \in \mathcal{I}})$ be a TCW. The matching relation $(\cap^I)_{I \in \mathcal{I}}$ is used in two contexts: (i) while connecting a clock reset point (say for clock $x$) to a point where a guard of the form $x \in I$ is checked, and (ii) while connecting a point where a push was made to its corresponding pop, where the age of the topmost stack symbol is checked to be in interval $I$.

We use the notations $\cap^x$ and $\cap^e$ to denote the matching relation $\cap^I$ corresponding to a clock-reset-check as well as push on stack-check respectively. We say that $\mathcal{V}$ is well timed w.r.t. a set of clocks $X$ and a stack $s$ if for each interval $I \in \mathcal{I}$ the matching relation $\cap^I$ can be partitioned as $\cap^I = \cap^x \cup \mathcal{V} \cap^e$ where

$(T_1)$ the stack relation $\cap^x = \bigcup_{I \in \mathcal{I}} \cap^x$ corresponds to the matching push-pop events, hence it is well-nested: for all $i \cap^x j$ and $i' \cap^x j'$, if $i < i' < j$ then $j' < j$.

$(T_2)$ For each $x \in X$, the clock relation $\cap^x = \bigcup_{I \in \mathcal{I}} \cap^x$ corresponds to the timing constraints for clock $x$ and respects the last reset condition: for all $i \cap^x j$ and $i' \cap^x j'$, if $i < i'$, then $j \leq j'$. See Figure 1 for example, where $0 \cap^x 2$ and $2 \cap^x 3$. 

![Figure 1](image-url)
It is then easy to check that TCWs defined by a TPDA with set of clocks $X$ are well-timed for the set of clocks $X$, i.e., satisfy the properties above. We obtain the same for TA by just ignoring the stack edges, i.e., $(T_1)$ above.

## 3 Tree-Width for Timed Systems

In this section, we discuss tree-algebra by introducing the basic terms, the operations on terms, their syntax and semantics. This will help us in analyzing the graphs obtained in the previous section using tree-terms, and establishing a bound on the tree-width.

We introduce tree terms $TTs$ from Courcelle [10] and their semantics as graphs which are both vertex-labeled and edge-labeled. Let $\Sigma$ be a set of vertex labels and let $\Xi$ be a set of edge labels. Let $K \in \mathbb{N}$. The syntax of $K$-tree terms $K$-$TTs$ over $(\Sigma, \Xi)$ is given by

$$\tau::= (a, i) \mid (a, i)\xi(b, j) \mid \text{Forget}_{i,}\tau \mid \text{Rename}_{i,j}\tau \mid \tau \oplus \tau$$

where $i, j \in \{1, 2, \ldots, K\}$ are colors $(i \neq j)$, $a, b \in \Sigma$ are vertex labels and $\xi \in \Xi$ is an edge label. The semantics of a $K$-$TT$ $\tau$ is a colored graph $[\tau] = (G_\tau, \chi_\tau)$ where $G_\tau = (V, E)$ is a graph and $\chi_\tau: \{1, 2, \ldots, K\} \rightarrow V$ is a partial injective function assigning a color to some vertices of $G_\tau$. Note that any color in $\{1, 2, \ldots, K\}$ is assigned to at most one vertex of $G_\tau$.

The atomic term $(a, i)$ is a single vertex colored $i$ and labeled $a$ and the atomic term $(a, i)\xi(b, j)$ represents a $\xi$-labeled edge between two vertices colored $i, j$ and labeled $a, b$ respectively. Given a tree term $\tau$, $\text{Forget}_{i}(\tau)$ forgets the color $i$ from a node colored $i$, leaving it uncolored. The operation $\text{Rename}_{i,j}(\tau)$ renames the color $i$ of a node to color $j$, provided no nodes are already colored $j$. Since any color appears at most once in $G_\tau$, the operations $\text{Forget}_{i}(\tau)$ and $\text{Rename}_{i,j}(\tau)$ are deterministic, when colors $i, j$ are fixed. Finally, the operation $\tau_1 \oplus \tau_2$ (read as combine) combines two terms $\tau_1, \tau_2$ by fusing the nodes of $\tau_1, \tau_2$ which have the same color. See Figure 2.

The tree-width of a graph $G$ is defined as the least $K$ such that $G = G_\tau$ for some $TT$ $\tau$ using $K + 1$ colors. Let $TW_K$ denote the set of all graphs having tree width at most $K$. For TCWs, we have successor edges $\rightarrow$ and matching edges $\cap I$ where $I \in \mathcal{I}$ is an interval. Hence, the set of edge labels is $\Xi_\mathcal{I} = \{\rightarrow\} \cup \{\cap I \mid I \in \mathcal{I}\}$ and we use $TT$s over $(\Sigma, \Xi_\mathcal{I})$. An example is given in [5, Appendix A].

**TCWs and Games.** We find it convenient to prove that TCWs have bounded tree-width by playing a game, whose game positions are TCWs in which some successor edges may have been cut, i.e., are missing. Such TCWs, where some successor edges may be missing, are called split-TCWs. A split-TCW which is a connected graph is called a connected split-TCW, while a split-TCW which is a disconnected graph, is called a disconnected split-TCW. For example, $\cdots$ is a connected split-TCW, while $\cdots$ is a disconnected split-TCW consisting of two connected split TCWs, namely $\cdots$ and $\cdots$.

A $TCW$ is atomic if it is denoted by an atomic term $((a, i) \rightarrow (b, j))$ or $(a, i) \cap I (b, j))$. The split-game is a two player turn based game $G = (\text{Pos}_{\exists}, \text{Pos}_{\forall}, \text{Moves})$ where Eve's

\[\text{CONCUR 2017}\]
Towards an Efficient Tree Automata Based Technique for Timed Systems

set of game positions $\text{Pos}_\exists$ consists of all connected (wrt. $\to \cup \setminus$) split-TCWs and Adam’s set of game positions $\text{Pos}_\forall$ consists of dis-connected split-TCWs. Eve’s moves consist of adding colors to the vertices of the split-TCW, and dividing the split-TCW. For example, if we have the connected split-TCW $\xrightarrow{\text{---}}$, and Eve colors two nodes (we use shapes in place of colors for better visibility) we obtain $\xrightarrow{\text{---}}$ and $\xrightarrow{\text{---}}$. This graph can be divided obtaining the disconnected graph $\xrightarrow{\text{---}}$ and $\xrightarrow{\text{---}}$. Now Adam’s choices are on this disconnected split-TCW and he can choose either of the above three connected split-TCWs to continue the game. Thus, divide is the reverse of the combine operation $\oplus$. Adam’s moves amount to choosing a connected component of the split-TCW. Eve has to continue coloring and dividing on the connected split-TCW chosen by Adam. Atomic split-TCWs are terminal positions in the game: neither Eve nor Adam can move from an atomic split-TCW. A play on a split-TCW $V$ is a path in $G$ starting from $V$ and leading to an atomic split-TCW. The cost of the play is the maximum width (number of colors-1) of any split-TCW encountered in the path. In our example above, $\xrightarrow{\text{---}}$ is already an atomic split-TCW. If Adam chooses any of the other two, it is easy to see that Eve has a strategy using at most 2 colors in any of the split-TCWs that will be obtained till termination. The cost of a strategy $\sigma$ for Eve from a split-TCW $V$ is the maximal cost of the plays starting from $V$ and following strategy $\sigma$. The tree-width of a (split-)TCW $V$ is the minimal cost of Eve’s (positional) strategies starting from $V$. Let $\text{TCW}_K$ denote the set of TCWs with tree-width bounded by $K$.

A block in a split-TCW is a maximal set of points of $V$ connected by $\to$. For example, the split-TCW $\xrightarrow{\text{---}} \xrightarrow{\text{---}}$ has one non-trivial block $\xrightarrow{\text{---}}$ and one trivial block $\xrightarrow{\text{---}}$. Points that are not left or right endpoints of blocks of $V$ are called internal.

The Bound. We show that we can find a $K$ such that all the behaviors of the given timed system have tree-width bounded by $K$.

▶ Theorem 3. Given a timed system $S$ using a set of clocks $X$, all graphs in its TCW language have tree-width bounded by $K$, i.e., $\text{TCW}(S) \subseteq \text{TCW}_K$, where
1. $K = |X| + 1$ if $S$ is a timed automaton,
2. $K = 3|X| + 2$ if $S$ is a timed pushdown automaton.

The following lemma completes the proof of Theorem 3 (2).

▶ Lemma 4. The tree-width of a well-timed TCW is bounded by $3|X| + 2$.

We prove this by playing the “split game” between Adam and Eve in which Eve has a strategy to disconnect the word without introducing more than $3|X| + 3$ colors. Eve’s strategy processes the word from right to left. Starting from any TCW, Eve colors the end points of the TCW, as well as the last reset points (from the right end) corresponding to each clock. Here she uses at most $|X| + 2$ colors. On top of this, depending on the last point, we have different cases, as sketched below (a detailed proof is in [5, Appendix B]).

If the last point is the target of a $\xrightarrow{x}$ edge for some clock $x$, then Eve simply removes the clock edge, since both the source and target points of this edge are colored. We only discuss in some detail the case when the last point is the target of a $\xleftarrow{\text{---}}$ edge, and the source of this edge is an internal point in the non-trivial block. Figure 3 illustrates this case.

To keep a bound on the number of colors needed, Eve divides the TCW as follows:
- First Eve adds a color to the source of the stack edge
- If there are any clock edges crossing this stack edge, Eve adds colors to the corresponding reset points. Note that this results in adding atmost $|X|$ colors.
Figure 3 The last point is the target of a $\rightarrow^*$ (top figure). After the split, we obtain the words $V_1$ (the middle one) and $V_2$ (the bottom one).

Eve disconnects the TCW into two parts, such that the right part $V_2$ consists of one non-trivial block whose end points are the source and target points of the stack edge, and also contains to the left of this block, at most $|X|$ trivial blocks. Each of these trivial blocks are the reset points of those clock edges which cross over. The left part $V_1$ is a TCW consisting of all points to the left of the source of the stack edge, and has all remaining edges other than the clock edges which have crossed over. Adam can now continue the game choosing $V_1$ or $V_2$. Note that in one of the words so obtained, the stack edge completely spans the non-trivial block, and can be easily removed.

**Invariants and bound on tree-width.** We now discuss some invariants on the structure of the split-TCWs as we play the game using the above strategy.

1. We have $\leq |X|$ colored trivial blocks to the left of the only non-trivial block,
2. The last reset node of each clock on the non-trivial block is colored,
3. The end points of the non-trivial block are colored.

To maintain the above invariants, we need $|X| + 1$ extra colors than the at most $2|X| + 2$ mentioned above. This proves that the tree-width of a TPDA with set of clocks $X$ is bounded by $3|X| + 2$. If the underlying system is a timed automaton, then we have a single non-trivial block in the game at any point of time. There are no trivial blocks, unlike the TPDA, due to the absence of stack edges. This results in using only $\leq |X| + 2$ colors at any point of time, where $|X|$ colors are needed to color the last reset points of the clocks in the block, and the 2 colors are used to color the left, right end points of the block.

### 4 Tree automata for Validity

In this section, we give one of the most challenging constructions (Theorem 6) of the paper, namely, the tree automaton that accepts all valid and realizable $K$-TTs which are "good". Good $K$-TTs are defined below. In this section, we restrict ourselves to closed intervals; that is, those of the form $[a, b]$ and $[a, \infty)$, where $a, b \in \mathbb{N}$. Fix $K \geq 2$. Not all graphs defined by $K$-TTs are realizable TCWs. Indeed, if $\tau$ is such a TT, the edge relation $\rightarrow$ may have cycles or may be branching, which is not possible in a TCW. Also, the timing constraints given by $\wedge^I$ need not comply with the $\rightarrow$ relation: for instance, we may have a timing constraint $e \wedge^I f$ with $f \rightarrow^+ e$ ($\rightarrow^+$ is the transitive closure of $\rightarrow$, i.e., $e$ can be reached from $f$ after taking $\geq 1$ successor edges $\rightarrow$). Moreover, some terms may define graphs denoting TCWs which are not realizable. So we use $A_{\text{valid}}^{K,M}$ to check for validity. Since we have only closed intervals in timing constraints, integer timestamps suffice for realizability, as can be seen from the following lemma ([5, Appendix C.1]).
Towards an Efficient Tree Automata Based Technique for Timed Systems

Table 1 The second row gives tree representations of three good 6-TTs $\tau_1, \tau_2, \tau_3$. In all these terms, we ignore vertex labels and we use $\text{Add}_{i,j}^a$ as a macro for $\tau \oplus i \bowtie^1 j$. The third row gives their semantics $[\tau] = (G_\tau, \chi_\tau)$ together with a realization $ts$, the fourth row gives possible states $q$ of $A_{\text{valid}}^{K,M}$ with $M = 4$ after reading the terms. Here, $L$ is the circled color. The boolean value $\text{acc}(i)$ for each non maximal color $i$ is written between $tsm(i)$ and $tsm(i^*)$.

<table>
<thead>
<tr>
<th></th>
<th>$\tau_1$</th>
<th>$\tau_2$</th>
<th>$\tau_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\text{Add}_{3,5}^{[2,\infty]}$</td>
<td>$\text{Add}_{2,6}^{[5,\infty]}$</td>
<td>$\text{Forget}_{5}$</td>
</tr>
<tr>
<td></td>
<td>$\text{Add}_{4,5}^{[1,3]}$</td>
<td>$\text{Add}_{4,6}^{[1,3]}$</td>
<td>$\text{Rename}_{3,4}$</td>
</tr>
<tr>
<td></td>
<td>$3 \rightarrow 4$</td>
<td>$5 \rightarrow 6$</td>
<td>$\text{Forget}_{5}$</td>
</tr>
<tr>
<td></td>
<td>$\chi$</td>
<td>$\chi$</td>
<td>$\text{ts}$</td>
</tr>
<tr>
<td>ts</td>
<td>1 3 4 5</td>
<td>2 3 4 5</td>
<td>1 2 3 4 6</td>
</tr>
<tr>
<td>P</td>
<td>$\neg \text{acc} \ 4 \ 5$</td>
<td>$\text{acc} \ 3 \ 4 \ 5$</td>
<td>$\text{tsm} \ 0 \ 3 \ 5 \ 6 \ 8 \ 11$</td>
</tr>
<tr>
<td>tsm</td>
<td>$\neg \text{acc} \ 1 \ 2 \ 3 \ 4 \ 5 \ 6$</td>
<td>$\text{acc} \ 0 \ 3 \ 4 \ 5 \ 6 \ 8 \ 11$</td>
<td>$\text{tsm} \ 0 \ 3 \ 5 \ 6 \ 8 \ 11$</td>
</tr>
</tbody>
</table>

Lemma 5. Let $\mathcal{V} = (V, \rightarrow, \lambda, (\bowtie^I)_{I \in \mathbb{Z}^{(M)}})$ be a TCW using only closed intervals in its timing constraints. Then, $\mathcal{V}$ is realizable iff there exists an integer valued timestamp map $\lambda$ satisfying all timing constraints.

Consider a set of colors $P \subseteq \{1, \ldots, K\}$. For each $i \in P$ we let $i^+ = \min\{j \in P \cup \{\infty\} \mid i < j\}$ and $i^- = \max\{j \in P \cup \{0\} \mid j < i\}$. If $P$ is not clear from the context, then we write $\text{prev}_P(i)$ and $\text{next}_P(i)$. Given a $K$-TT $\tau$ with semantics $[\tau] = (G, \chi)$, we denote by $\text{Act} = \text{dom}(\chi)$ the set of active colors in $\tau$, we let $\text{Right} = \max(\text{Act})$ and $\text{Left} = \min\{i \in \text{Act} \mid \chi(i) \rightarrow^* \chi(\text{Right})\}$. If $\tau$ is not clear from the context, then we write $\text{Act}_\tau$, $\text{Left}_\tau$, and $\text{Right}_\tau$. A $K$-TT $\tau$ is good if

- $\tau ::= (a, i) \rightarrow (b, j) \mid (a, i) \bowtie^1 (b, j) \mid \text{Forget}_{i,j}(\tau) \mid \text{Rename}_{i,j}(\tau) \mid \tau \oplus \tau$,
- for every subterm of the form $(a, i) \rightarrow (b, j)$ or $(a, i) \bowtie^1 (b, j)$ we have $i < j$,
- $\text{Rename}_{i,j}(\tau)$ is possible only if $i^- < j < i^+$,
- $\tau_1 \oplus \tau_2$ is allowed if $\text{Right}_{\tau_1} = \text{Left}_{\tau_2}$ and $\{i \in \text{Act}_{\tau_2} \mid \text{Left}_{\tau_1} \leq i \leq \text{Right}_{\tau_1}\} \subseteq \text{Act}_{\tau_1}$.

Examples of good TTs and their semantics are given in Table 1. Note that the semantics of a $K$-TT $\tau$ is a colored graph $[\tau] = (G_\tau, \chi_\tau)$. Below, we provide a direct construction of a tree automaton, which gives a clear upper bound on the size of $A_{\text{valid}}^{K,M}$, since obtaining this bound gets very technical if we stick to MSO.

Theorem 6. We can build a tree automaton $A_{\text{valid}}^{K,M}$ with $M^{O(K)}$ number of states such that $L(A_{\text{valid}}^{K,M})$ is the set of good $K$-TTs $\tau$ such that $[\tau]$ is a realizable TCW and the endpoints of $[\tau]$ are the only colored points.

Proof. The tree automaton $A_{\text{valid}}^{K,M}$ reads the TT bottom-up and stores in its state a finite abstraction of the associated graph. The finite abstraction will keep only the colored points of the graph. We will only accept good terms for which the natural order on the active colors...
coincides with the order of the corresponding vertices in the final TCW. The restriction to good terms ensures that the graph defined by the TT is a split-TCW.

Moreover, to ensure realizability of the TCW defined by a term, we will guess timestamps of vertices modulo $M$. We also guess while reading a subterm whether the time elapsed between two consecutive active colors is big ($\geq M$) or small ($< M$). We see below that the time elapsed is small iff it can be recovered accurately with the modulo $M$ abstraction. Then, the automaton has to check that all these guesses are coherent and using these values the time elapsed is $A(M(i)) = (D(i), tsm)$. Blocks of a split-

<table>
<thead>
<tr>
<th>Table 2 Transitions of $A_{\text{valid}}^{K,M}$</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$(a,i) \to (b,j)$</td>
<td>$q = (P, L, tsm, acc)$ is a transition if $i &lt; j$ and $P = {i, j}$, $L = i$ and $acc(j) = \text{ff}$. The values for $tsm(i)$, $tsm(j)$ and $acc(i)$ are guessed.</td>
</tr>
<tr>
<td>$(a,i) \cap \not\rightarrow (b,j)$</td>
<td>$q = (P, L, tsm, acc)$ is a transition if $i &lt; j$ and $P = {i, j}$, $L = j$ and $acc(j) = \text{ff}$. Here, $i$ and $j$ are trivial blocks. The values for $tsm(i)$, $tsm(j)$ and $acc(i)$ are guessed such that $(acc(i) = \text{tt} \land d(i,j) \in I) \lor (acc(i) = \text{ff} \land I.\text{up} = \infty)$.</td>
</tr>
<tr>
<td>Rename$_{i,j}$</td>
<td>$q = (P, L, tsm, acc)$ is a transition if $i &lt; j$ and $i &lt; j &lt; i^*$. Then, $q'$ is obtained from $q$ by replacing $i$ by $j$.</td>
</tr>
<tr>
<td>Forget$_{i,j}$</td>
<td>$q = (P, L, tsm, acc)$ is a transition if $L &lt; i &lt; \max(P)$ (endpoints should stay colored). Then, state $q'$ is deterministically given by $P' = P \setminus {i}$, $L' = L$, $tsm' = tsm_{P'}$ and $acc'(i') = \text{ACC}(i', i^<em>) \land (D(i', i^</em>) &lt; M)$, the other values of $acc'$ are inherited from $acc$.</td>
</tr>
<tr>
<td>$\oplus$</td>
<td>$q_1, q_2 \to q$ where $q_1 = (P_1, L_1, tsm_1, acc_1)$, $q_2 = (P_2, L_2, tsm_2, acc_2)$ and $q = (P, L, tsm, acc)$ is a transition if the following hold:</td>
</tr>
<tr>
<td>$R_1 = \max(P_1) = L_2$ and $(i \in P_2 \setminus L_1 \mid L_1 \leq i \leq R_1) \subseteq P_1$ (we cannot insert a new point from the second argument in the non-trivial block of the first argument).</td>
<td></td>
</tr>
<tr>
<td>$P = P_1 \cup P_2$, $L = L_1$, and $tsm_{P_1} = tsm_1$ and $tsm_{P_2} = tsm_2$: these updates are deterministic. In particular, this implies that $tsm_1$ and $tsm_2$ coincide on $P_1 \cap P_2$.</td>
<td></td>
</tr>
</tbody>
</table>
| Finally, $acc$ satisfies $acc(\max(P))$ if and only if $\forall i \in P_1 \setminus \max(P_1)$ $acc_1(i) \iff \text{ACC}(i, next_{P_1}(i)) \land D_1(i, next_{P_1}(i)) < M$ $\forall i \in P_2 \setminus \max(P_2)$ $acc_2(i) \iff \text{ACC}(i, next_{P_2}(i)) \land D_2(i, next_{P_2}(i)) < M$. Notice that these conditions imply:
| For all $i \in P_1$, if $next_{P_1}(i) = next_{P_1}(i)$ (e.g., if $L_1 \leq i < R_1$) then $acc(i) = acc_1(i)$. |
| For all $i \in P_2$, if $next_{P_2}(i) = next_{P_2}(i)$ (e.g., if $L_2 \leq i$) then $acc(i) = acc_2(i)$. |

Intuitively, when reading bottom-up a $K$-TT $\tau$ with $[\tau] = (V, \rightarrow, \lambda, (\wedge^J)_{I \in \mathbb{Z}}, \chi)$, the automaton $A_{\text{valid}}^{K,M}$ will reach a state $q = (P, L, tsm, acc)$ such that

$(A_1)$ $P = \text{Act}$ is the set of active colors in $\tau$, $L = \text{Left}$ and $\max(P) = \text{Right}$. 

$(A_2)$ For all $i \in P$, if $L \leq i < \max(P)$ then $\chi(i) \to^+ \chi(i^*)$ in $[\tau]$. 

$(A_3)$ Let $\rightarrow^* = \{(\chi(i), \chi(i^*)) \mid i \in P \land i < L\}$. This extra relation serves at ordering the blocks of a split-TCW. Then, $[\tau]$, $\rightarrow^*$ is an ordered split-TCW, i.e., $< = (\to \cup \rightarrow^*)^+$ is a total order on $V$, timing constraints in $[\tau]$ are $<\text{-compatible}$ $\cap I \subseteq \leq$ for all $I$, the direct successor relation of $< \,$ is $< \,$ $\to \cup \rightarrow^*$ and $\to \cap \rightarrow^* = \emptyset$. Moreover, targets of timing constraints are in the last block: for all $u \cap I^J \ni v$ in $[\tau]$, $\rightarrow^*$, we have $\chi(L) \to^* v$. 

CONCUR 2017
Towards an Efficient Tree Automata Based Technique for Timed Systems

(A4) There exists a timestamp map $\text{ts}: V \to \mathbb{N}$ such that
- all constraints are satisfied: $\text{ts}(u) - \text{ts}(v) \in I$ for all $u \cap^J v$ in $\llbracket \tau \rrbracket$,
- time is non-decreasing: $\text{ts}(u) \leq \text{ts}(v)$ for all $u \leq v$,
- $(\text{tsm}, \text{acc})$ is the modulo $M$ abstraction of $\text{ts}$: $\forall i \in P$ we have $\text{tsm}(i) = \text{ts}(\chi(i))[M]$ and $\text{acc}(i) = \text{tt}$ if $i^+ \neq \infty$ and $\text{ts}(\chi(i))^+ - \text{ts}(\chi(i)) < M$.

We say that the state $q$ is a realizable abstraction of a term $\tau$ if it satisfies conditions (A1)–(A4).

Indeed, the finite state automaton $A^{K,M}_{\text{valid}}$ cannot store the timestamp map $\text{ts}$ witnessing realizability. Instead, it stores the modulo $M$ abstraction $(\text{tsm}, \text{acc})$. We will see that $A^{K,M}_{\text{valid}}$ can check realizability based on the abstraction $(\text{tsm}, \text{acc})$ of $\text{ts}$ and can maintain this abstraction while reading the term bottom-up.

We introduce some notations. Let $q = (P, L, \text{tsm}, \text{acc})$ be a state and let $i, j \in P$ with $i \leq j$. We define $d(i, j) = (\text{tsm}(j) - \text{tsm}(i))[M]$ and $D(i, j) = \sum_{k \in P \mid i \leq k < j} d(k, k^+)$. We also define $\text{ACC}(i, j) = \bigwedge_{k \in P \mid i \leq k < j} \text{acc}(k)$. If the state is not clear from the context, then we write $d_q(i, j), D_q(i, j), \text{ACC}_q(i, j)$. For instance, with the state $q_3$ corresponding to the term $\tau_3$ of Table 1, we have $\text{ACC}(1, 4) = \text{tt}$, $d(1, 4) = 2$ and $D(1, 4) = 6 = \text{ts}(4) - \text{ts}(1)$ is the accurate value of the time elapsed. Whereas, $\text{ACC}(3, 6) = \text{ff}$ and $d(3, 6) = 2 = D(3, 6)$ are both strict modulo-$M$ under-approximations of the time elapsed $\text{ts}(6) - \text{ts}(3) = 6$. The transitions of $A^{K,M}_{\text{valid}}$ are defined in Table 2.

**Accepting condition.** The accepting states of $A^{K,M}_{\text{valid}}$ should correspond to abstractions of TCWs. Hence the accepting states are of the form $(\{i, j\}, L, \text{tsm}, \text{acc})$ with $i, j \in \{1, \ldots, K\}$, $i < j$, $L = i$ and $\text{acc}(j) = \text{ff}$. The correctness of this construction is in [5, Appendix C.2], and is obtained by proving (i) the transitions of $A^{K,M}_{\text{valid}}$ indeed preserve the conditions (A1)–(A4), (ii) (A1)–(A4) ensure among other things, that the boolean values $\text{acc}(i)$, $\text{ACC}(i, j)$ for $i < j$ indeed defines when the elapse of time is accurately captured by the modulo $M$ abstraction: that is, $\text{ACC}(i, j)$ is true iff the actual time elapse between $i$ and $j$ is captured using the modulo $M$ abstraction $D(i, j)$.

## 5 Tree automata for timed systems

The goal of this section is to build a tree automaton which accepts tree terms denoting TCWs accepted by a TPDA. The existence of a tree automaton can be proved by showing the MSO definability of the runs of the TPDA $S$ on a TCW. However, as seen in section 4, we directly construct a tree automaton for better complexity. Given the timed system $S$, let $K$ be the bound on tree-width given by Theorem 3 and let $M$ be one more than the maximal constant occurring in the guards of $S$. The automaton $A^{K,M}_{S}$ will accept good $K$-TTs with the additional restriction that a timing constraint is immediately combined with an existing term. That is, restricted $K$-TTs are good $K$-TTs restricted to the following syntax:

$$\tau ::= (a,i) \to (b,j) \mid \tau \oplus [(a,i) \cap^I (b,j)] \mid \text{Forget}_{i,j} \tau \mid \text{Rename}_{i,j} \tau \mid \tau \oplus \tau$$

**Theorem 7.** Let $S$ be a TPDA of size $|S|$ (constants encoded in unary) with set of clocks $X$ and using constants less than $M$. Let $K$ be the bound on tree-width given by Theorem 3. Then, we can build a tree automaton $A^{K,M}_{S}$ with $|S|^{O(K)} \cdot K^{O(|X|)}$ states such that $A^{K,M}_{S}$ accepts the set of restricted $K$-TTs $\tau$ such that $[\tau] \in \text{TCW}(S)$. Further, $\text{TCW}(S) = \llbracket L(A^{K,M}_{S}) \rrbracket = \{[\tau] \mid \tau \in L(A^{K,M}_{S})\}$.\hfill $\blacksquare$
Proof (Sketch). A state of $\mathcal{A}^{K,M}_2$ is a tuple $q = (P, L, \delta, \text{Push}, \text{Pop}, G, Z)$ where,

- $P$ is the set of active colors, and $L = \text{Left} \in P$ is the left-most point that is connected to the right-end-point $R = \text{Right} = \max(P)$ by successor edges on the non-trivial block.
- $\delta$ is a map that assigns to each color $k \in P$ the transition $\delta(k)$ guessed at the leaf corresponding to color $k$.
- Push and Pop are two boolean variables: Push = 1 iff a push-pop edge has been added to $L$ and Pop = 1 iff a push-pop edge has been added to $R$.
- $G = (G_x)_{x \in X}$ is a boolean vector of size $|X|$: for each clock $x \in X$, $G_x = 1$ iff some constraint on $x$ has already been checked at $R$.
- $Z = (Z_x)_{x \in X}$ assigns to each clock $x$ either the color $i \in P$ with $i < L$ of the unique point on the left of the non-trivial block which is the source of a timing constraint $i \cap^j j$ for clock $x$, or $\perp$ if no such points exist.

For $j \in P$, let $\text{Reset}(j)$ be the set of clocks that are reset in the transition $\delta(j)$. We describe here the most involved kind of transition $q' \oplus q''$ for states $q', q''$. The remaining transitions as well as the full proof are in [5, Appendix D]. Let $q' = (P', L', \delta', \text{Push}', \text{Pop}', G', Z')$, $q'' = (P'', L'', \delta'', \text{Push}'', \text{Pop}'', G'', Z'')$ and $q = (P, L, \delta, \text{Push}, \text{Pop}, G, Z)$. Then $q', q'' \overset{\oplus}{\rightarrow} q$ is a transition if the following hold:

\[ C_1: \ R' = \max(P') = L'' \text{ and } \{i \in P'' \mid L' \leq i \leq R''\} \subseteq P' \ (\text{we cannot insert a new point from the second argument in the non-trivial block of the first argument}). \]

\[ \text{Note that according to } C_4, \text{ the points } \bullet, \bigtriangleup \text{ and } \blacklozenge \text{ in } P'' \text{ lying between } L', R' \text{ are already points in the non-trivial block connecting } L' \text{ to } R'. \]

\[ C_2: \ \forall i \in P'' \cap P', \delta(i) = \delta''(i) \ (\text{the guessed transitions match}). \]

\[ \text{By } C_2, \text{ the transitions } \delta', \delta'' \text{ of } \bullet, \bigtriangleup \text{ and } \blacklozenge \text{ must match.} \]

\[ C_3: \text{ if there is a Push operation in } \delta''(L'') \text{ then } \text{Push}'' = 1 \text{ and if there is a pop operation in } \delta'(R') \text{ then } \text{Pop} = 1 \ (\text{the push-pop edges corresponding to the merging point have been added, if they exist}). \]

\[ \text{By } C_3, \text{ if } \delta(R') = \delta(L'') \text{ contains a pop (resp. push) operation then } R' = L'' \text{ is the target (resp. source) of a push-pop edge.} \]

\[ C_4: \text{ if some guard } x \in I \text{ is in } \delta(R'), \text{ then } G_x' = 1 \ (\text{before we merge, we ensure that the clock guard for } x \text{ in the transition guessed at } R', \text{ if any, has been checked}). \]

\[ \text{After the merge, } R' = L'' \text{ becomes an internal point; hence by } C_4, \text{ any guard } x \in I \text{ in } \delta'(R') \text{ must be checked already, i.e., } G_x' = 1. \]

\[ \text{After the merge, it is no more possible to add an edge } \cap^j \text{ leading into } R'. \]

\[ C_5: \text{ if } Z_x' \neq \perp, \text{ then } \forall j \in P'', \ Z_x' < j < L' \text{ implies } x \notin \text{Reset}''(j) \ (\text{If a matching edge starting at } Z_x' < L' \text{ had been seen earlier in run leading to } q', \text{ then } x \text{ should not have been reset in } q' \text{ between } Z_x' \text{ and } L', \text{ else it would violate the consistency of clocks}). \]

\[ \text{By } C_5, \text{ if } Z_x' \text{ is } \bullet \text{ (resp. } \blacklozenge), \text{ i.e., } \bigtriangleup \text{ (resp. } \bullet \text{)} \text{ is the source of a timing constraint } \cap^j \text{ for clock } x \text{ whose target is in the } L'-R' \text{ block, then clock } x \text{ cannot be reset at } \bigtriangleup \text{ and } \bullet \text{ (resp. } \blacklozenge). \]

\[ C_6: \text{ if } Z_x' \neq \perp, \text{ then } \forall j \in P', \ Z_x'' < j < L'' \text{ implies } x \notin \text{Reset}'(j) \ (\text{If a matching edge starting at } Z_x'' < L'' \text{ had been seen earlier in run leading to } q'', \text{ then } x \text{ should not have been reset in } q' \text{ between } Z_x'' \text{ and } L''). \]

\[ \text{By } C_6, \text{ if } Z_x'' \text{ is } \blacklozenge, \text{ then } x \text{ cannot be reset at } \blacklozenge, \bullet, \bigtriangleup, \text{ or } \blacklozenge. \]

Likewise, if $Z_x''$ was $\bullet$, then clock $x$ cannot be reset at $\blacklozenge, \bullet, \bigtriangleup, \text{ or } \blacklozenge$.\]

CONCUR 2017
Towards an Efficient Tree Automata Based Technique for Timed Systems

\[ C_T: \ P = P' \cup P'', \ L = L', \ \delta = \delta' \cup \delta'', \ \text{Push} = \text{Push}', \ \text{Pop} = \text{Pop}'' \text{ and } G = G'' \]

for all \( x \in X \) we have \( Z_x = Z_x''\) if \( Z_x'' < L'\), else \( Z_x = Z_x'\). \( C_T \) says that on merging, we obtain the third split-TCW. After the merge, if \( Z_x \) is defined, it must be on the left of \( L' \), i.e., one of \( \diamond, \blacklozenge, \bigcirc, \triangleright \). Notice that the above three conditions ensure the well-nestedness of clocks. By \( C_5 \) and \( C_6 \) we cannot have both \( Z_x' \in \{ \diamond, \blacklozenge \} \) and \( Z_x'' \in \{ \blacklozenge, \bigcirc, \triangleright \} \). So if \( Z_x'' \in \{ \diamond, \blacklozenge \} \) then \( Z_x = Z_x'' \) and otherwise \( Z_x = Z_x' \) (including when \( Z_x'' \in \{ \blacklozenge, \bigcirc, \triangleright \} \) and \( Z_x' = \bot \)).

**Accepting Condition.** A state \( q = (P, L, \delta, \text{Push}, \text{Pop}, G, Z) \) is accepting if \( L = \min(P) \), \( \delta(L) \) is some dummy \( \varepsilon \)-transition resetting all clocks and leading to the initial state, \( \text{target}(\delta(R)) \) is a final state and if \( \delta(R) \) has a pop operation then \( \text{Pop} = 1 \), if it has a constraint/guard for clock \( x \), then \( G_x = 1 \). Note that the above automaton only accepts restricted \( K \)-TTs; this is sufficient for emptiness checking since Eve’s winning strategy in Section 3 captures all behaviours of the TCW(\( S \)) while generating only restricted \( K \)-TTs. \( \triangleright \)

As a corollary we obtain (see [5, Appendix D.2]),

**Theorem 8.** Let \( S \) be a TPDA. We have \( L(S) \neq \emptyset \) iff \( L(\mathcal{A}_{\text{valid}}^{K,M} \cap \mathcal{A}_{S}^{K,M}) \neq \emptyset \).

If the underlying system is a timed automaton, we can restrict the state space to storing just the tuple \( (P, \delta, G) \) as the other components are not required and \( L \) is always \( \min(P) \).

**Possible Extensions.** We now briefly explain how to extend our technique in the presence of diagonal guards: these are guards of the form \( x - y \in I \) or \( \text{pop} - x \in I \) where \( x, y \) are clocks, and \( I \) is a time interval. The former is a guard that checks the difference between two clock values, while the latter checks the difference between the value of a clock and the age of the topmost stack symbol at the time of the pop. To handle a constraint of the form \( x - y \in I \), it is enough to check the difference between the guessed time stamps at the last reset points of clocks \( y, x \) to be in \( I \). Likewise, to check \( \text{pop} - x \in I \) or \( \text{pop} - x \in I \), we check the difference between the guessed time stamps at the points where the top symbol was pushed on the stack and the last reset of clock \( x \). Note that the last reset points of \( x, y \) will not be forgotten until the automaton decides to accept; likewise, the push point will not be forgotten until the pop transition is encountered. Given this, our construction of the tree automaton can be extended with the above checks to handle diagonal guards as well.

### 6 Implementation and a case-study

We have implemented the emptiness checking procedure for TPDA using our tree-automata based approach, and describe some results here. Despite the EXPTIME-completeness of this problem for general TPDA, we present some good performance results for certain interesting subclasses of TPDA. As a concrete subclass, the complexity significantly improves when there is no extra clock other than the timing constraints associated with the stack; while popping a symbol, we simply check the time elapsed since the push. Note that this can be used to model systems where timing constraints are well-nested: clock resets correspond to push and checking guards corresponds to checking the age of the topmost stack symbol. Thus, this gives a technique for reducing the number of clocks for a timed system with nested timing constraints. For this subclass, the exact number of states of the tree automaton can be improved to \( 2 \times (M \times T)^2 \), where \( M \) is 1 plus the maximum constant, and \( T \) is the number of transitions. This idea can be extended further to incorporate clocks whose constraints
are well-nested with respect to the stack. We can also handle clocks which are reset and checked in consecutive transitions.

For the general model (one stack + any number of clocks), we can use optimizations to reduce the number of states of the tree automaton to $\left( M \times T \right)^{2|X|+2} \times 2^{2|X|+1}$, where $|X|$ is the number of clocks, $M$ is 1 plus the maximum constant and $T$ is the number of transitions. To see this, consider the worst case scenario, where a state of the tree automaton has $|X|$ hanging points and $|X|$ reset points. In total there can be $2|X| + 2$ active points including the left and right end-points of the non-trivial block. After a combine operation, we can forget a point $i$ of the new state, if it is the case that every clock $x$ reset at the transition (guessed) at point $i$ is also reset at some transition at a point after $i$. Following this strategy, if we aggressively forget as many points as we can, we will have at most $|X|$ internal (reset) active points between the left and right end-points of the non-trivial block. Thus, we reduce the number of active points from $3|X| + 2$ to $2|X| + 2$.

As a proof of concept, we have implemented our approach with these optimizations. We will now describe some examples we modelled and their experimental results. These experiments were run on a 3.5 GHz i5 PC with 8GB RAM, with number of cores=4.

A Modeling Example : Maze with Constraints

As an interesting example, we model a situation of a robot successfully traversing a maze respecting multiple constraints (see Figure 4). These constraints may include logical constraints: the robot must visit location 1 before exit, or the robot must load something at a certain place $i$ and unload it at place $j$ (so number of visits to $i$ must equal visits to $j$). We may also have local and global time constraints which check whether adjacent places are visited within a time bound, or the total time taken in the maze is within a given duration. We show below, via an illustrative example, that certain classes of such constraints can be converted into a 1-clock TPDA.

One can go from place $p$ to some of its adjacent place $q$ if there is an arrow from place $p$ to place $q$. In addition, the following types of constraints must be respected.

1. **Logical constraints** specify certain order between visiting places, the number of times (upper/lower bounds) to visit a place or places, and so on. The logical constraints we have in our example are (a) place 1 must be visited exactly once, (b) from the time we enter the maze, to visiting place 1, one must visit place 7 (load) and place 4 (unload) equal number of times, and at any point of time, the number of visits to place 7 is not less than number of visits to place 4. (c) from visiting place 1 to exiting the maze, one
Towards an Efficient Tree Automata Based Technique for Timed Systems

must visit place 7 and place 4 equal number of times and, at any point during time, number of visits to place 7 is not less than number of visits to place 4.

2. **Local time constraints** specify time intervals which must be respected while going from a place to its adjacent place. The time taken from some place $i$ to an adjacent place $j$ is given as a closed interval $[a, b]$ along with the arrow. One cannot spend any time between a pair of adjacent places other than the ones specified in the maze. For example, the time bound for going from place 7 to 3 is given, while the time taken from place 3 to place 7 and place 6 to place 1 is zero $([0,0])$, since it is not mentioned. Further, one cannot stay in any place for non-zero duration.

3. **Global time constraints** specify total time that can elapse between visiting two places. From entering the maze to visiting of place 1, time taken should be exactly $m$ units (a parameter). From visiting place 1 to exit, time should be exactly $n$ units (another parameter).

A maze respecting multiple constraints as above is converted into a 1-clock TPDA. While the details of this conversion are given in [5, Appendix E], the main idea is to encode local time bounds with the clock which is reset on all transitions. A logical constraint specifying equal number of visits to places $p_1, p_2$ is modelled by pushing symbols while at $p_1$, and popping them at $p_2$. Likewise, if there is a global time constraint that requires a time elapse in $[a, b]$ between the entry and some place $p$, then push on the stack at entry, and check its age while at $p$. Note that these are well-nested properties.

To check the existence of a legitimate path in the maze respecting the constraints, our tool checks the existence of a run in the TPDA. By running our tool on the TPDA constructed (and fixing the parameters to be $m = 7$, $n = 8$), we obtain the following run: (described as a sequence of pairs the form : State, Entry time stamp in the state) $(6, 0.0) \rightarrow (3, 0.0) \rightarrow (7, 0.0) \rightarrow (3, 1.0) \rightarrow (7, 1.0) \rightarrow (3, 2.0) \rightarrow (5, 5.0) \rightarrow (4, 5.0) \rightarrow (5, 6.0) \rightarrow (4, 6.0) \rightarrow (5, 7.0) \rightarrow (6, 7.0) \rightarrow (1, 7.0) \rightarrow (6, 7.0) \rightarrow (3, 7.0) \rightarrow (7, 7.0) \rightarrow (3, 9.0) \rightarrow (7, 9.0) \rightarrow (3, 10.0) \rightarrow (5, 13.0) \rightarrow (4, 13.0) \rightarrow (5, 14.0) \rightarrow (4, 14.0) \rightarrow (5, 15.0) \rightarrow (6, 15.0) \rightarrow (2, 15.0)$.

The scalability is assessed by instantiating the maze for various choices of maximum constants used, as well as number of transitions. The running times with respect to various choices for the maximum constant are plotted on the right. More examples can be found in [5, Appendix E].

### 7 Conclusion

We have obtained a new construction for the emptiness checking of TPDA, using tree-width. The earlier approaches [1], [2] which handle dense time and discrete time push down systems respectively use an adaptation of the well-known idea of timed regions. The technique in [2] does not extend to dense time systems, and it is not clear whether the approach in [1] will work for say, multi stack push down automata even with bounded scope/phase restrictions. Unlike this, our approach is uniform: all our proofs except the tree automaton for realizability already work even if we have open guards. Our realizability proof has to be adapted for open guards and this is work under progress; in this paper, we focussed on closed guards to obtain an efficient tool based on our theory. Likewise,
our proofs can be extended to bounded phase/scope/rounds multi stack timed push down automata: we need to show a bound on the tree-width, and then adapt the tree automaton construction for the system automaton. The tree automaton checking realizability requires no change. With the theoretical improvements in this paper, we implement our approach and examine its performance on real examples. To the best of our knowledge, this is the first tool implementing timed push down systems. We plan to optimize our implementation to get a more robust and scalable tool. For the subclasses we have, it would be good to have a characterization and automatic translation (currently done by hand) that replaces well-nested clock constraints by stack edges, thus leading to better implementability.

Acknowledgments. The authors thank Vincent Jugé for insightful discussions on the MSO definability of realizability of TC words.

References

On Petri Nets with Hierarchical Special Arcs

S. Akshay¹, Supratik Chakraborty², Ankush Das³, Vishal Jagannath⁴, and Sai Sandeep⁵

¹ Department of Computer Science and Engineering, IIT Bombay, India
² Department of Computer Science and Engineering, IIT Bombay, India
³ Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, USA
⁴ Department of Computer Science and Engineering, IIT Bombay, India
⁵ Department of Computer Science and Engineering, IIT Bombay, India

Abstract

We investigate the decidability of termination, reachability, coverability and deadlock-freeness of Petri nets endowed with a hierarchy of places, and with inhibitor arcs, reset arcs and transfer arcs that respect this hierarchy. We also investigate what happens when we have a mix of these special arcs, some of which respect the hierarchy, while others do not. We settle the decidability status of the above four problems for all combinations of hierarchy, inhibitor, reset and transfer arcs, except the termination problem for two combinations. For both these combinations, we show that deciding termination is as hard as deciding the positivity problem for linear recurrence sequences – a long-standing open problem.

1998 ACM Subject Classification
F.1.1 Models of Computation, D.2.2 Design Tools and Techniques: Petri nets

Keywords and phrases Petri Nets, Hierarchy, Reachability, Coverability, Termination, Positivity

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2017.40

1 Introduction

Petri nets are an important and versatile mathematical modeling formalism for distributed and concurrent systems. Thanks to their intuitive visual representation, precise execution semantics, well-developed mathematical theory and availability of tools for reasoning about them, Petri nets are used in varied contexts, viz. computational, chemical, biological, workflow-related etc. Several extensions to Petri nets have been proposed in the literature to augment their modeling power. From a theoretical perspective, these provide rich and interesting models of computation that warrant investigation of their expressive powers, and decidability and/or complexity of various decision problems. From a practitioner’s perspective, they enable new classes of systems to be modeled and reasoned about.

In this paper, we focus on an important class of extensions proposed earlier for Petri nets, pertaining to the addition of three types of special arcs, namely inhibitor, reset and transfer arcs from places to transitions. We investigate how different combinations of these extensions affect the decidability of four key decision problems: reachability, coverability, termination, and deadlock-freeness. To start with, an inhibitor arc effectively models a zero test, and hence one can model two-counter machines with two inhibitor arcs, leading to undecidability of all of the above decision problems. However, Reinhardt [20] showed that if we impose a hierarchy among places with inhibitor arcs (a single inhibitor arc being a sub-case), we recover decidability of reachability. Recently Bonnet [5] simplified this proof using techniques of Leroux [14] and also showed that termination and coverability are decidable.
for Petri nets with hierarchical inhibitor arcs. With reset arcs (which remove all tokens from a pre-place) and transfer arcs (which transfer all tokens from a pre-place to a post-place), reachability and deadlock-freeness are known to be undecidable [9], although termination and coverability are decidable [11].

In this paper, we are interested in what happens when hierarchy is introduced among all combinations of special arcs. Thus, we specify a hierarchy, or total ordering, of the places, and say that the special arcs respect the hierarchy if whenever there is a special arc from a place $p$ to a transition $t$, there are also special arcs from every place lower than $p$ in the hierarchy to $t$. The study of Petri nets extended with hierarchical and non-hierarchical special arcs provides a generic framework that subsumes several existing questions and raises new ones. There are only a handful of results in the literature where hierarchical special arcs have been shown to play an important role. Decidability of reachability for Petri nets with hierarchical inhibitor arcs was shown in [20] and re-visited in a special context in [4], while decidability of termination, coverability and boundedness were shown in [5]. Further, in [2] it was shown that Petri nets with hierarchical zero tests are equivalent to Petri nets with a stack encoding restricted context-free languages. Finally a specific subclass, namely Petri nets with a single inhibitor arc, has received a lot of attention, with results showing decidability of boundedness and termination [10], place-boundedness [6], and LTL model checking [7]. However, in [7], the authors remark that it would not be easy to extend their technique for the last two problems to handle hierarchical arcs. To the best of our knowledge, none of the earlier papers address mixing of reset and transfer arcs within the hierarchy of inhibitor arcs, leaving several interesting questions unanswered. Our primary goal in this paper is to comprehensively fill these gaps. Before delving into the theoretical investigations, we present two examples that illustrate why these models are interesting from a practical point of view.

Our first example is a prioritized job-shop environment in which work stations with possibly different resources are available for servicing jobs. Each job comes with a priority and a requirement of the count of resources it needs. For simplicity, assume that all resources are identical, and that there is at most one job with any given priority. A work station can service multiple jobs simultaneously subject to availability of resources; however, a job cannot be split across multiple work stations. Additionally, we require that a job with a lower priority must not be scheduled on any work station as long as a job with higher priority is waiting to be scheduled. Once a job gets done, it can either terminate or generate additional jobs with different priorities based on some rules. An example of such a rule could be that a job with priority $k$ and resource requirement $m$ can only generate a new job with priority $\leq k$ and resource requirement $\leq m$. Given such a system, there are several interesting questions one might ask. For example, can too many jobs (above a specified threshold) of the lowest priority be left waiting for a work station? Or, can the system reach a deadlocked state from where no progress can be made? A possible approach to answering these questions is to model this prioritized job-shop environment as a Petri net with hierarchical special arcs, i.e., resets, inhibitors and transfers, and reduce the questions to decision problems (such as coverability or deadlock-freeness) for the corresponding nets.

Our second example builds on work reported in the literature on modeling integer programs with loops using Petri nets [3]. Questions pertaining to termination of such programs can be reduced to decision problems (termination or deadlock-freeness) of the corresponding Petri net model. In Section 6.1, we describe a new reduction of the termination question for integer linear loop programs to the termination problem for Petri nets with hierarchical inhibitor and transfer arcs. This underlines the importance of studying decision problems for these extensions of Petri nets.
Our main contribution is a comprehensive investigation into Petri nets extended with a mix of these special arcs, some of which respect the hierarchy, while others do not. We settle the decidability status of the four decisions problems for all combinations of hierarchy, inhibitor, reset and transfer arcs, except the termination problem for two combinations. For these cases, we show a reduction from the positivity problem [18, 19] – a long-standing open problem on linear recurrences. We summarize these results in Section 3, after introducing appropriate notations in Section 2. Interestingly, several of our results use completely different constructions and proof techniques, as detailed in Sections 4–6.

2 Preliminaries

We begin by recalling some key definitions and fixing notations. A Petri net, denoted $\mathcal{PN}$, is defined as $(P, T, F, M_0)$, where $P$ is a set of places, $T$ is a set of transitions, $M_0 : P \to \mathbb{N}$ is the initial marking, and $F : (P \times T) \cup (T \times P) \to \mathbb{N}$ is the flow relation. For every $x \in P \cup T$, we define $\text{Pre}(x) = \{ y \in P \cup T \mid F(y, x) > 0 \}$ and $\text{Post}(x) = \{ y \in P \cup T \mid F(x, y) > 0 \}$. For every $t \in T$, we use the following terminology: every $p \in \text{Pre}(t)$ is a pre-place of $t$, every $q \in \text{Post}(t)$ is a post-place of $t$, every arc $(p, t)$ such that $F(p, t) > 0$ is a pre-arc of $t$, and every arc $(t, p)$ such that $F(t, p) > 0$ is a post-arc of $t$.

A marking $M : P \to \mathbb{N}$ is a function from the set of places to non-negative integers. We say that a transition $t$ is firable at marking $M$, denoted by $M \overset{t}{\rightarrow}$, if $\forall p \in \text{Pre}(t), M(p) \geq F(p, t)$. If $t$ is firable at $M_1$, we say that firing $t$ gives the marking $M_2$, where $\forall p \in P, M_2(p) = M_1(p) - F(p, t) + F(t, p)$. This is also denoted as $M_1 \overset{t}{\rightarrow} M_2$. We define the sequence of transitions $\rho = t_1t_2...t_n$ to be a run from marking $M_0$, if there exist markings $M_1, M_2, ..., M_n$, such that for all $i$, $t_i$ is firable at $M_{i-1}$ and $M_{i-1} \overset{t_i}{\rightarrow} M_i$. Finally, we abuse notation and use $\preceq$ to denote the component-wise ordering over markings. Thus, $M_1 \preceq M_2$ if $\forall p \in P, M_1(p) \leq M_2(p)$. A detailed account on Petri nets can be found in [17].

We now define some classical decision problems in the study of Petri nets.

- **Definition 2.1.** Given a Petri net $N = (P, T, F, M_0)$,
  - **Termination (or TERM):** Does there exist an infinite run from marking $M_0$?
  - **Reachability (or REACH):** Given a marking $M$, is there a run from $M_0$ which reaches $M$?
  - **Coverability (or COVER):** Given a marking $M$, is there a marking $M' \geq M$ which is reachable from $M_0$?
  - **Deadlock-freeness (or DLFREE):** Does there exist a marking $M$ reachable from $M_0$, such that no transition is firable at $M$?

Since Petri nets are well-structured transition systems (WSTS), the decidability of coverability and termination for Petri nets follows from the corresponding results for WSTS [11]. The decidability of reachability was shown in [13]. Subsequently, there have been several alternative proofs of the same result, viz. [14]. Finally, since deadlock-freeness reduces to reachability in Petri nets [8], all the four decision problems are decidable for Petri nets. In the remainder of the paper, we concern ourselves with these decision problems for Petri nets extended with the following special arcs:

- An **Inhibitor arc** from place $p$ to transition $t$ signifies $t$ is firable only if $p$ has zero tokens.
- A **Reset arc** from place $p$ to transition $t$ signifies that $p$ contains zero tokens after $t$ fires.
- A **Transfer arc** from place $p_1$ through transition $t$ to place $p_2$ signifies that on firing transition $t$, all tokens from $p_1$ get transferred to $p_2$.

For Petri nets with special arcs, we redefine the flow relation as $F : (P \times T) \cup (T \times P) \to \mathbb{N} \cup \{ I, R \} \cup \{ S_p \mid p \in P \}$, where $F(p, t) = I$ (resp. $F(p, t) = R$) signifies the presence of an
inhibitor arc (resp. reset arc) from place \( p \) to transition \( t \). Similarly, if \( F(p, t) = S_p' \), then there is a transfer arc from place \( p \) to place \( p' \) through transition \( t \).

3 Problem statements and main results

We use \( PN \) to denote standard Petri nets, and \( I-PN \), \( R-PN \) and \( T-PN \) to denote Petri nets with inhibitor, reset and transfer arcs, respectively. The following definition subsumes several additional extensions studied in this paper.

- **Definition 3.1.** A Petri net with **hierarchical special arcs** is defined to be a 5-tuple \((P, T, F, \sqsubseteq, M_0)\), where \( P \) is a set of places, \( T \) is a set of transitions, \( \sqsubseteq \) is a total ordering over \( P \) encoding the hierarchy, \( M_0 : P \to \mathbb{N} \) is the initial marking, and \( F : (T \times P) \cup (P \times P) \to \mathbb{N} \sqcup \{I, R\} \sqcup \{S_p | p \in P\} \) is a flow relation satisfying:

\[
\forall (t, p) \in T \times P, \quad F(t, p) \in \mathbb{N}, \quad \text{and} \quad \forall (p, t) \in P \times T, \quad F(p, t) \not\in \mathbb{N} \implies (\forall q \sqsubseteq p, \quad F(q, t) \not\in \mathbb{N})
\]

Thus, all arcs (or edges) from transitions to places are as in standard Petri nets. However, we may have special arcs from places to transitions. These can be inhibitor arcs \((F(p, t) = I)\), reset arcs \((F(p, t) = R)\), or transfer arcs \((F(p, t) = S_p')\), where \( p \) and \( p' \) are places in the Petri net. Note that all special arcs respect the hierarchy specified by \( \sqsubseteq \). In other words, if there is a special arc from a place \( p \) to a transition \( t \), there must also be special arcs from every place \( p' \) to \( t \), where \( p' \sqsubseteq p \). Depending on the subset of special arcs that are present, we can define sub-classes of Petri nets with hierarchical special arcs as follows. In the following, \( \text{Range}(F) \) denotes the range of the flow relation \( F \).

- **Definition 3.2.** The class of Petri nets with hierarchical special arcs, where \( \text{Range}(F) \setminus \mathbb{N} \) is a subset of \( \{I\}, \{T\} \) or \( \{R\} \) is called \( \text{HIPN} \), \( \text{HTPN} \) or \( \text{HRPN} \) respectively. Similarly, it is called \( \text{HIITPN} \), \( \text{HIRPN} \) or \( \text{HTRPN} \) if \( \text{Range}(F) \setminus \mathbb{N} \) is a subset of \( \{I, T\}, \{I, R\} \) or \( \{T, R\} \) respectively. Finally, if \( \text{Range}(F) \setminus \mathbb{N} \subseteq \{I, R, T\} \), we call the corresponding class \( \text{HIRTPN} \).

We also study generalizations, in which extra inhibitor, reset and/or transfer arcs that do not respect the hierarchy specified by \( \sqsubseteq \), are added to \( PN \) with hierarchical special arcs.

- **Definition 3.3.** Let \( \mathcal{N} \) be a class of Petri nets with hierarchical special arcs as in Definition 3.2, and let \( \mathcal{M} \) be a subset of \( \{I, T, R\} \). We use \( \mathcal{M}-\mathcal{N} \) to denote the class of nets obtained by adding unrestricted special arcs of type \( \mathcal{M} \) to an underlying net in the class \( \mathcal{N} \).

For example, \( R-\text{HIPN} \) is the class of Petri nets with hierarchical inhibitor arcs extended with reset arcs that need not respect the hierarchy. Clearly, if the special arcs in every net \( N \in \mathcal{N} \) are from \( \mathcal{M} \), the class \( \mathcal{M}-\mathcal{N} \) is simply the class of Petri nets with unrestricted (no hierarchy) arcs of type \( \mathcal{M} \). Hence we avoid discussing such extensions in the remainder of the paper.

As we show later, all four decision problems of interest to us are either undecidable or not known to be decidable for \( \text{HIRTPN} \). A slightly constrained version of \( \text{HIRTPN} \), however, turns out to be much better behaved, motivating the following definition.

- **Definition 3.4.** The sub-class \( \text{HIRcTPN} \) is defined to be \( \text{HIRTPN} \) with the added restriction that \( \forall (p, t, p') \in P \times T \times P, \quad F(p, t) = S_p' \implies F(p', t) \in \mathbb{N} \).

Thus, every place \( p' \) that has an incoming transfer arc through transition \( t \) is constrained to have only a standard \( PN \) outgoing arc (if any) to \( t \). This restriction suffices to recover decidability for all four decision problems of interest to us. Since nets in \( \text{HIRcTPN} \) often suffice to model useful classes of systems, we present results for this class separately.
Table 1 Summary of key results; results for all other extensions are subsumed by these results.

<table>
<thead>
<tr>
<th>Term</th>
<th>Cover</th>
<th>Reach</th>
<th>DLFree</th>
</tr>
</thead>
<tbody>
<tr>
<td>PN</td>
<td>✓ [11]</td>
<td>✓ [11]</td>
<td>✓ [15, 14]</td>
</tr>
<tr>
<td>I-PN</td>
<td>✓ [16]</td>
<td>✓ [16]</td>
<td>✓ [16]</td>
</tr>
<tr>
<td>HIPN</td>
<td>✓ [20, 5]</td>
<td>✓ [20, 5]</td>
<td>✓ [20, 5]</td>
</tr>
<tr>
<td>HTPN</td>
<td>✓ [Thm 4.2]</td>
<td>✓ [Thm 4.2]</td>
<td>✓ [Thm 4.2]</td>
</tr>
<tr>
<td>HITPN</td>
<td>Positivity-Hard [Thm 6.5]</td>
<td>✓ [Cor. 6.2]</td>
<td>✓ [Thm 6.1]</td>
</tr>
<tr>
<td>HIRcTPN</td>
<td>✓ [Thm 4.2]</td>
<td>✓ [Thm 4.2]</td>
<td>✓ [Thm 4.2]</td>
</tr>
<tr>
<td>R-HIPN</td>
<td>✓ [Thm 5.1]</td>
<td>✓ [Thm 5.6]</td>
<td>✓ [9], [1]</td>
</tr>
<tr>
<td>T-HIPN</td>
<td>Positivity-Hard [Thm 6.5]</td>
<td>✓ [Thm 5.6]</td>
<td>✓ [9], [1]</td>
</tr>
<tr>
<td>R-HIRPN</td>
<td>✓ [Thm 5.1, Thm 4.2]</td>
<td>✓ [Thm 5.6]</td>
<td>✓ [9], [1]</td>
</tr>
</tbody>
</table>

Status of decision problems and our contributions. Table 1 summarizes the decidability status of the four decision problems for some classes of Petri net extensions. A ✓ denotes decidability of the corresponding problem, while × denotes undecidability of the problem. The shaded cells present results (and corresponding citations) already known prior to the current work, while the unshaded cells show results (and corresponding theorems) arising from this paper. Note that the table doesn’t list all extensions of Petri nets that were defined above. This has been done deliberately and carefully to improve readability. Specifically, for every Petri net extension that is not represented in the table, e.g., R-HITPN, the status of all four decision problems are inferable from others shown in the table. These are explicitly listed out in a longer version of the paper [1], where we also depict the relative expressiveness of these classes. Thus, our work comprehensively addresses the four decision problems for all classes of Petri net extensions considered above.

Interestingly, several of the results use distinct constructions and proof techniques. We point out the salient features of our main results below.

- **We include reset arcs in the hierarchy** of inhibitor arcs in HIPN in Section 4. In Theorem 4.2, we show that we can model reset arcs by inhibitor arcs, while preserving hierarchy. This immediately gives decidability of all problems except DLFree. Since the reduction may introduce deadlocks, we need a different proof for DLFree, which we present through in Theorem 4.3. This also proves decidability of DLFree for HIPN, which to the best of our knowledge, was not known before.

- **We add reset arcs outside the hierarchy** of inhibitor arcs in Section 5. Somewhat counter-intuitively, this class, R-HIPN, does not contain HIRPN and is incomparable to it. This is because all inhibitor arcs in R-HIPN are required to respect the hierarchy, whereas in HIRPN some inhibitor arcs can be replaced by resets, thereby violating the requirement of hierarchy among inhibitor arcs. Using a new and surprisingly simple construction of an extended finite reachability tree (FRT) which keeps track of the hierarchical inhibitor information and modifies the subsumption condition, we show in Theorem 5.1 that termination is decidable for R-HIPN. This result has many consequences. In particular, it implies an arguably simple proof for the special case of a single inhibitor arc, which was solved in [10] (using a different method of extending FRTs). In Theorem 5.6, we use a reduction from two counter machines to show that coverability is undecidable with as few as 2 reset arcs and an inhibitor arc in the absence of hierarchy.

- **Finally, we consider transfer arcs inside and outside the hierarchy** in Section 6. In Theorem 6.1, we show that unlike for reset arcs, including transfer arcs in the hierarchy...
of inhibitor arcs does not necessarily preserve decidability. For both \textsc{HITPN} and \textsc{T-HIPN}, while coverability, reachability and deadlock-freeness are undecidable, we are unable to show such a result for termination. Instead, in Theorem 6.5, we show that we can reduce a long-standing open problem on linear recurrences to this problem.

4 Adding Reset Arcs with Hierarchy to HIPN

In this section, we extend hierarchical inhibitor nets [20] with reset arcs respecting hierarchy. Subsection 4.1 presents a reduction from \textsc{HIRPN} to \textsc{HIPN} that settles the decidability of termination, coverability and reachability for \textsc{HIRPN}. As this reduction does not work for deadlock-freeness (since it introduces new deadlocked markings), we present a separate reduction from deadlock-freeness to reachability for \textsc{HIRPN} in subsection 4.2.

4.1 Reduction from HIRPN to HIPN

Let \textsc{HIRPN}\(_k\) be the sub-class of Petri nets in \textsc{HIRPN} with at most \(k\) transitions having one or more reset pre-arcs. We first show that termination, reachability and coverability for \textsc{HIRPN}\(_k\) can be reduced to the corresponding problems for \textsc{HIRPN}\(_{k-1}\), for all \(k > 0\). This effectively reduces these problems for \textsc{HIRPN} to the corresponding problems for \textsc{HIRPN}\(_0\) (or \textsc{HIPN}), which are known to be decidable [20, 5]. In the following, we use \textsf{Markings}(\(N\)) to denote the set of all markings of a net \(N\).

Lemma 4.1. For every net \(N\) in \textsc{HIRPN}\(_k\), there is a net \(N'\) in \textsc{HIRPN}\(_{k-1}\) and a mapping \(f : \textsf{Markings}(\(N\)) \rightarrow \textsf{Markings}(\(N'\))\) that satisfy the following:

- For every \(M_1, M_2 \in \textsf{Markings}(\(N\))\) such that \(M_2\) is reachable from \(M_1\) in \(N\), the marking \(f(M_2)\) is reachable from \(f(M_1)\) in \(N'\).
- For every \(M'_1, M'_2 \in \textsf{Markings}(\(N'\))\) such that \(M'_1 = f(M_1), M'_2 = f(M_2)\) and \(M'_2\) is reachable from \(M'_1\) in \(N'\), the marking \(M_2\) is reachable from \(M_1\) in \(N\).

Proof Sketch. To see how \(N'\) is constructed, consider an arbitrary transition, say \(t\), in \(N\) with one or more reset pre-arcs. We replace \(t\) by a gadget in \(N'\) with no reset arcs, as shown in Figure 1. The gadget has two new places labeled \(p^*\) and \(p^*_t\), with every transition in “Rest of Net” having a simple pre-arc from and a post-arc to \(p^*\), as shown by the dotted arrows in Figure 1. The gadget also has a new transition \(t^R\) with simple pre-arcs from \(p^*\) and from every place \(p^S\) that has a simple arc to \(t\) in \(N\). It also has a new transition labeled \(t^R\) for every reset arc from a place \(p^R\) to \(t\) in \(N\). Thus, if there are \(n\) reset pre-arcs of \(t\) in \(N\), the gadget has \(n\) transitions \(t^R_1, \ldots, t^R_n\). As shown in Figure 1, each such \(t^R_i\) has simple pre-arcs.
from \( p_i^R \) and \( p_i^* \) and a post-arc to \( p_i^* \). Finally, the gadget has a new transition labeled \( t^I \) with a simple pre-arc from \( p_i^* \) and inhibitor pre-arcs from all places \( p^I \) (resp. \( p^R \)) that have inhibitor (resp. reset) arcs to \( t \) in \( N \).

The ordering \( \sqsubseteq ' \) of places in \( N' \) is obtained by extending the ordering \( \sqsubseteq \) of \( N \) as follows: for each place \( p \) in \( N \), we have \( p \sqsubseteq ' p_i^* \sqsubseteq ' p^* \). Clearly, \( N' \in HIRcTPN_{k-1} \), since it has one less transition (i.e. \( t \)) with reset pre-arcs compared to \( N \). It is easy to check that if the reset and inhibitor arcs in \( N \) respect \( \sqsubset \), then the reset and inhibitor arcs in \( N' \) respect \( \sqsubseteq ' \) as well.

The mapping function \( f : \text{Markings}(N) \rightarrow \text{Markings}(N') \) is defined as follows: for every place \( p \) in \( N' \), \( f(M)(p) = M(p) \) if \( p \) is in \( N \); otherwise, \( f(M)(p^*) = 1 \) and \( f(M)(p_i^*) = 0 \). The initial marking of \( N' \) is given by \( f(M_0) \), where \( M_0 \) is the initial marking of \( N \). Given a run in \( N' \), it is now easy to see that every occurrence of \( t \) in the run can be replaced by the sequence \( t^S(t^R)^*t^I \) (the \( t^R \) transitions fire until the corresponding place \( p^R \) is emptied) and vice-versa. Further details of the construction are given in [1], where it is also shown that \( N' \) can reach \( M_2 \) from \( M_1 \) iff \( N' \) can reach \( f(M_2) \) from \( f(M_1) \).

In fact, the above construction can be easily adapted for HIRcTPN as well. Specifically, if we have a transfer arc from place \( p_x \) to place \( p_y \) through \( t \), we add a new transition \( t^I_{x,y} \) with simple pre-arcs from \( p_i^* \) and \( p_x \), and with simple post-arcs to \( p_i^* \) and \( p_y \) to the gadget shown in Figure 1. Furthermore, we add an inhibitor arc from \( p_x \) to \( t^I \), like the arc from \( p^R \) to \( t^I \) in Figure 1. Note that the constrained property of transfer arcs is required here, since if we had an inhibitor arc from \( p_y \) to \( t \) in original net (hence, \( p_y \) to \( t^I \) in construction), then in the constructed net, \( t^I \) cannot be fired, since we would have added tokens in \( p_y \) through \( t^T_{x,y} \). This allows us to obtain a net in HIRcTPN with at least one less transition with reset pre-arcs or transfer arcs, such that the reachability guarantees in Lemma 4.1 hold.

Finally, by repeatedly applying Lemma 4.1, we have,

\[ \textbf{Theorem 4.2.} \text{ Termination, reachability and coverability for HIRPN and HIRcTPN are decidable.} \]

### 4.2 Reducing Deadlock-freeness to Reachability in HIRPN

The overall idea behind our reduction is to add transitions that check whether the net is deadlocked, and to put a token in a special place, say \( p^* \), if this is indeed the case. Note that for a net to be deadlocked, the firing of each of its transitions must be disabled. Intuitively, if \( M \) denotes a marking of a net and \( I \) denotes the set of transitions, then \( \text{Deadlock}(M) = \bigwedge_{i \in I} \text{NotFire}_i(M) \), where \( \text{Deadlock}(M) \) is a predicate indicating if the net is deadlocked in \( M \), and \( \text{NotFire}_i(M) \) is a formula representing the enabledness of transition \( t_i \) in \( M \). For a transition \( t \) to be disabled, at least one of its pre-places \( p \) must fail the condition on that place for \( t \) to fire. There are three cases to consider here.

\[ \begin{align*}
F(p, t) & \in \mathbb{N}: \text{For } t \text{ to be disabled, we must have } M(p) < F(p, t) \\
F(p, t) & = 1: \text{For } t \text{ to be disabled, we must have } M(p) > 0. \\
F(p, t) & = R: \text{Place } p \text{ cannot disable } t
\end{align*} \]

Suppose we define \( \text{Exact}_j(p) \equiv (M(p) = j) \) and \( \text{AtLeast}(p) \equiv (M(p) > 0) \). Clearly, \( \text{NotFire}_i(M) = \bigvee_{(p,t) \in I} \text{Check}(p) \), where \( \text{Check}(p) = \text{AtLeast}(p) \) if \( F(p, t) = 1 \), and \( \text{Check}(p) = \bigvee_{j < k} \text{Exact}_j(p) \) if \( F(p, t) = k \in \mathbb{N} \). The formula for \( \text{Deadlock}(M) \) (in CNF above) is now converted into DNF by distributing conjunctions over disjunctions. Given a HIRPN, we now transform the net, preserving hierarchy, so as to reduce checking \( \text{Deadlock}(M) \) in DNF in the original net to a reachability problem in the transformed HIRPN.

Every conjunctive clause in the DNF of \( \text{Deadlock}(M) \) is a conjunction of literals of the form \( \text{AtLeast}(p) \) and \( \text{Exact}_j(p) \). Let \( S_C \) be the set of all literals in a conjunctive clause \( C \),
and let \( P \) be the set of all places in the net. Define \( B^C = \{ p \in P \mid \text{Exact}_i(p) \in S_C \} \) and \( A^C = \{ p \in P \mid \text{AtLeast}(p) \in S_C \} \setminus \bigcup_{i \geq 1} B^C_i \). We only need to consider conjunctive clauses where the sets \( B^C \) are pairwise disjoint (other clauses can never be true). Similarly, we only need to consider conjunctive clauses where \( B^C_0 \) and \( A^C \) are disjoint. We add a transition for each conjunctive clause that satisfies the above two properties. By definition, \( A^C \) and \( B^C_i \) are disjoint for all \( i \geq 1 \). Thus, the sets \( A^C \) and \( B^C_i \) (\( i \geq 0 \)) are pairwise disjoint for every conjunctive clause we consider.

![Diagram](image)

Given the original HIRPN net, for each conjunctive clause considered, we perform the construction as shown in the above figure. For every place \( p_a \in A^C \), we add a construction as for \( p_2 \) in above diagram. For every place \( p_i \in B^C_i \), we add a construction as for \( p_1 \) in above diagram. For all places \( p \notin A^C \cup \bigcup_i B^C_i \), we add a construction as for \( p_3 \) in above diagram. We call the transition \( t_C \) in the above diagram as the 'Check Transition', and refer to the set of transitions \( r_i, s_i, t_i, t_i\) (excluding \( q_C \)) as transitions for clause \( C \). Note that for any \( p_i \in P \), exactly one of \( r_i, s_i, t_i \) exist since the sets \( A^C \) and the sets \( B^C_i \) are all pairwise disjoint.

Our construction also adds two new places, \( p_C \) and \( p^{**} \), and one new transition \( q_C \) such that

- there is a pre-arc and a post-arc of weight 1 from \( p^{**} \) to every transition in the original net. Thus, transitions in original net can fire only if \( p^{**} \) has a token.
- there is a pre-arc of weight 1 from \( p_c \) to each transition for clause \( C \) (within dotted box).
- there is a post-arc of weight 1 to \( p_c \) from every transition for clause \( C \) (within dotted box), except from \( t_C \) to \( p_C \).

Note that hierarchy is preserved in the transformed net, since the only new transitions which have inhibitor/reset arcs are the check transitions, which have inhibitor arcs from all places in the original net. Let \( N \) be the original net in HIRPN with \( P \) being its set of places, and let \( N' \) be the transformed net, also in HIRPN, obtained above. Define a mapping \( f : \text{Markings}(N) \to \text{Markings}(N') \) as follows: \( f(M)(p) = M(p) \) if \( p \in P \); \( f(M)(p^{**}) = 1 \) and \( f(M)(p) = 0 \) in all other cases. If \( M_0 \) is the initial marking in \( N \), define \( M_0' = f(M_0) \) to be the initial marking in \( N' \).

**Claim 4.1.** The marking \( M'_s \) of \( N' \), defined as \( M'_s(p) = 1 \) if \( p = p^s \) and \( M'_s(p) = 0 \) otherwise, is reachable from \( M'_0 \) in \( N' \) iff there exists a deadlocked marking reachable from \( M_0 \) in \( N \).

From, the above reduction (see [1] for proof details) and from the decidability of reachability for HIRPN we then have the following.

**Theorem 4.3.** Deadlock-freeness for HIRPN is decidable.
5 Adding Reset Arcs without Hierarchy

The previous section dealt with extension of Petri nets where reset arcs were added within the hierarchy of the inhibitor arcs. This section discusses the decidability results when we add reset arcs outside the hierarchy of inhibitor arcs.

5.1 Termination in R-HIPN

Our main idea here is to use a modified finite reachability tree (FRT) construction to provide an algorithm for termination in R-HIPN. The usual FRT construction [11] for Petri nets does not extend to Petri nets with even a single (and hence also hierarchical) inhibitor arc.

\begin{theorem}
Termination is decidable for R-HIPN.
\end{theorem}

Consider a R-HIPN net \((P, T, F, \subseteq, M_0)\). We start by introducing a few definitions. For any place \(p \in P\), we define the index of the place \(p\) (\(Index(p)\)) as the number of places \(q \in P\) such that \(q \subseteq p\). The definition of Index over places induces an Index among transitions too: For any transition \(t \in T\), its index is defined as \(Index(t) = \max_{F(p, t) = 1} Index(p)\) by convention, if there is no such place, then \(Index(t) = 0\). Given markings \(M_1\) and \(M_2\) and \(i \in \mathbb{N}\), we say that \(M_1\) and \(M_2\) are \(i\)-Compatible (denoted \(Compat_i(M_1, M_2)\)) if \(\forall p \in P\), \((Index(p) \leq i \implies M_1(p) = M_2(p))\).

\begin{definition}
Consider a run \(M_2 \xrightarrow{\rho} M_1\). Let \(t^* = \arg\max_{t \in \rho} Index(t)\). We define \(Subsume(M_2, M_1, \rho) = M_2 \leq M_1 \land \left(Compat_{Index(t^*)}(M_1, M_2)\right)\).
\end{definition}

To understand this definition note that if \(\rho\) can be fired at \(M_2\) and reaches \(M_1\) and if \(Subsume(M_2, M_1, \rho)\) is true, then \(\rho\) can be fired at \(M_1\) again. In classical Petri nets without inhibitor arcs, \(Subsume(M_2, M_1, \rho) = M_2 \leq M_1\), and hence this is the classical monotonicity condition. However, this condition may differ in the presence of even a single inhibitor arc.

Given a net \(N = (P, T, F, \subseteq, M_0)\) in R-HIPN, we define the Extended Reachability Tree \(ERT(N)\) as a directed unordered tree where the nodes are labelled by markings \(M: P \rightarrow \mathbb{N}\), rooted at \(n_0 : M_0\) (initial marking). If \(M_1 \xrightarrow{t} M_2\) for some markings \(M_1\) and \(M_2\) and transition \(t \in T\), then a node marked by \(n' : M_2\) is a child of the node \(n : M_1\). Consider any node labelled \(M_1\). If along the path from root \(n_0 : M_0\) to \(n : M_1\), there is a marking \(n' : M_2\) (\(n \neq n'\)), such that the path from \(n' \rightarrow M_2\) to \(n : M_1\) corresponds to a run \(\rho\) and \(Subsume(M_2, M_1, \rho)\) is true, then \(M_1\) is made a leaf node. We call this a \emph{subsumed} leaf node. Note that leaf nodes in this tree are of two types: either leaf nodes caused by subsumption as above or leaf nodes due to deadlock, where no transition is fireable.

\begin{lemma}
For any net \(N = (P, T, F, \subseteq, M_0)\) in R-HIPN, \(ERT(N)\) is finite.
\end{lemma}

\begin{proof}
Assume the contrary. By Konig’s Lemma, there is an infinite path. Let the infinite path correspond to a run \(\rho = M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} M_2 \ldots \xrightarrow{t_i} M_j \ldots\).

Let \(t \in T\) be the transition which has maximum index among the transitions which are fired infinitely often in run \(\rho\). Thus all transitions having higher index than \(Index(t)\) fire only finitely many times. Let \(b\) be chosen such that \(\forall i \geq b\), \(Index(t_i) \leq Index(t)\) (i.e. \(b\) is chosen after the last position where any transition with index higher than \(Index(t)\) fires). This exists by the definition of \(t\). Since \(t\) is fired infinitely often, the sequence \(\{M_i\} i > b \land t_{i+1} = t\) is an infinite sequence. As \(\leq\) over markings is a well-quasi ordering, there exist two markings \(M_i\) and \(M_j\), such that both belong to the above sequence (i.e. \(t_{i+1} = t_{j+1} = t\), \(M_i \leq M_j\) and \(i < j\). Since \(t_{i+1} = t_{j+1} = t\), and \(t\) fires at \(M_i\) and \(M_j\), we have \(\forall p \in P\), \(Index(p) \leq \ldots\)
Index(t) \implies M_i(p) = M_j(p) = 0. Thus, \textit{Compat}_{\text{Index(t)}}(M_i, M_j) is true. Note that t is the maximum index transition fired in the run from \( M_i \) to \( M_j \), since no higher index transition fires after position \( b \) and \( j > i > b \). Hence, \( \text{Subsumed}(M_i, M_j, \rho') \) is true, where \( \rho' \) is the run from \( M_i \) to \( M_j \). But then, the path would end at \( M_j \), giving a contradiction. ◀

Thus, we have shown that the ERT is always finite. Next, we will show a crucial property of \( \text{Compat}_i \), which will allow us to check for a non-terminating run.

\begin{proof}
Consider markings \( M_1 \) and \( M_2 \) such that \( M_1 \leq M_2 \). Let \( i \in \mathbb{N} \) be such that we have \( \text{Compat}_i(M_1, M_2) \). Then for any run \( \rho \) over the set of transitions \( T_i = \{ t \mid t \in T \land \text{Index}(t) \leq i \} \), if \( M_1 \xrightarrow{\rho} M'_1 \), then \( M_2 \xrightarrow{\rho} M'_2 \), where \( M'_1 \leq M'_2 \) and \( \text{Compat}_i(M'_1, M'_2) \).

**Proof.** We prove this by induction. We first prove that \( t \) is fireable at \( M_2 \). If \( F(p, t) \in \mathbb{N} \), then \( M_2(p) \geq M_1(p) \geq F(p, t) \). If \( F(p, t) = 1 \), i.e., it is an inhibitor arc, then \( \text{Index}(p) \leq \text{Index}(t) \leq i \). But now, since \( \text{Compat}_i(M_1, M_2) \) holds and \( t \) is fireable at \( M_1 \), we obtain \( M_2(p) = M_1(p) = 0 \). Finally, if \( F(p, t) = R \), i.e., it is a reset arc, then \( t \) can fire regardless of the value of \( M_2(p) \). Hence, \( t \) is fireable at \( M_2 \).

Now let \( M_2 \xrightarrow{1} M'_2 \). Then, for all \( p \in P \), \( M'_2(p) = M_2(p) - F(p, t) + F(t, p) \), where \( F(t, p) \) is constant and \( F(p, t) \) can depend only on number of tokens in place \( p \) (so, if \( M_1(p) \) and \( M_2(p) \) were equal before firing, they remain equal now), we obtain that \( \text{Compat}_i(M'_1, M'_2) \) and \( M'_1 \leq M'_2 \). ◀

\begin{lemma}
A net \( N \) in R-HIPN has a non-terminating run iff ERT(\( N \)) has a subsumed leaf node.

**Proof.** In the forward direction, consider a non-terminating run. This run has a finite prefix in ERT(\( N \)). This prefix ends in a leaf that is not a deadlock (as some transition is fireable). Thus it is a subsumed leaf node. In the reverse direction, we now show that if ERT(\( N \)) has a subsumed leaf node, then \( N \) has a non-terminating run. To see this, consider any subsumed leaf node labeled by marking \( M_2 \). Let \( M_1 \) be a marking along the path \( M_0 \) to \( M_2 \), and \( \rho \) be the run from \( M_1 \) to \( M_2 \), such that \( \text{Subsumed}(M_2, M_1, \rho) \) is true. Hence, we have \( M_1 \xrightarrow{\rho} M_2 \). Take \( t^* = \arg\max_{t \in \text{Index}(t)} \text{Index}(t) \) and \( i = \text{Index}(t^*) \). Since \( \text{Subsumed}(M_1, M_2, \rho) \) is true, we have \( M_1 \leq M_2 \) and \( \text{Compat}_i(M_1, M_2) \) is true. We also have that \( \rho \) is a run over \( T_i = \{ t \mid t \in T \land \text{Index}(t) \leq i \} \) (by definition of \( i \)). Thus, by Lemma 5.4, we have \( M_2 \xrightarrow{\rho} M_3 \), where \( M_2 \leq M_3 \) and \( \text{Compat}_i(M_2, M_3) \) is true. Hence, \( \rho \) can be fired again at \( M_3 \) and so on, resulting in a non-terminating run. ◀

Finally, we observe that checking \( \text{Subsumed}(M_2, M_1, \rho) \) is also easily doable. Thus, for any net in R-HIPN, one can construct its extended reachability tree and decide the termination problem using the ERT. This completes the proof of the theorem. We observe here that this construction cannot be immediately lifted to checking boundedness due to the presence of reset arcs. However, we can lift this to check for termination in HIRPN and R-HIRPN as well as to check boundedness in HIPN.

### 5.2 Coverability in R-HIPN

While termination turned out to be decidable, reachability is undecidable for R-HIPN in general (since it subsumes reset Petri nets). Indeed, it is shown in [9] that reachability is undecidable for Petri nets with 2 reset arcs. Using a similar strategy, in [1], we tighten the undecidability result to show that reachability in Petri nets with one inhibitor arc and one
reset arc is undecidable. Further, we can modify the construction presented to show that deadlock-freeness in Petri nets with one reset arc and one inhibitor arc is also undecidable.

Next we turn our attention to the coverability problem.

\textbf{Theorem 5.6.} Coverability is undecidable for Petri nets with two reset/transfer arcs and an inhibitor arc.

The rest of this section proves the above theorem. To do this, we construct a Petri net with two reset arcs and one inhibitor arc that simulates a two-counter Minsky Machine. A Minsky Machine $M$ has a finite set of instructions $q_i$ for $0 \leq i \leq n$, where $q_0$ is the initial instruction and $q_n$ is the final instruction i.e. there are no transition rules from $q_n$. There are two counters $C_1$ and $C_2$ and we have two types of instructions. For each counter $r \in \{1, 2\}$,

1. \textsc{INC}($r, j$): Increase $C_r$ by 1 and go to $q_j$.
2. \textsc{JZDEC}($r, j, l$): If $C_r$ is zero, go to $q_l$, else decrease $C_r$ by 1 and go to $q_j$.

We construct a Petri net $P$ such that the runs of $P$ encode the runs of the Minsky Machine $M$. We use places $q_i$ ($0 \leq i \leq n$) to encode each instruction. The place $q_i$ gets a token when we simulate instruction $i$ in the Minsky Machine. We use two places $C_1$ and $C_2$ to store the number of tokens corresponding to counter values in $C_1$ and $C_2$ in the counter machine. We use a special place $S$ which stores the sum of $C_1$ and $C_2$. Figure 2(a) shows the construction for the increment instruction “Increase $C_r$ by 1 and go to $q_j$". When $q_i$ gets a token, the transition is fired, $C_r$ and $S$ are incremented by 1 and $q_j$ gets the token to proceed.

Next, we show how to simulate the decrement (along with zero check) instruction “if $C_r = 0$, then go to $q_l$, else decrease $C_r$ by 1 and go to $q_j$" by introducing non-determinism in the Petri net. The gadget for this is shown in Figure 2(b). When $q_i$ gets a token, the transition is fired, $C_r$ and $S$ are incremented by 1 and $q_j$ gets the token to proceed.

We construct a Petri net $P$ such that the runs of $P$ encode the runs of the Minsky Machine $M$. We use places $q_i$ ($0 \leq i \leq n$) to encode each instruction. The place $q_i$ gets a token when we simulate instruction $i$ in the Minsky Machine. We use two places $C_1$ and $C_2$ to store the number of tokens corresponding to counter values in $C_1$ and $C_2$ in the counter machine. We use a special place $S$ which stores the sum of $C_1$ and $C_2$. Figure 2(a) shows the construction for the increment instruction “Increase $C_r$ by 1 and go to $q_j$". When $q_i$ gets a token, the transition is fired, $C_r$ and $S$ are incremented by 1 and $q_j$ gets the token to proceed.

Next, we show how to simulate the decrement (along with zero check) instruction “if $C_r = 0$, then go to $q_l$, else decrease $C_r$ by 1 and go to $q_j$" by introducing non-determinism in the Petri net. The gadget for this is shown in Figure 2(b). When we reach a decrement with zero check instruction, we guess whether $C_r$ is zero, and if so, fire $t_{11}$ and then $t_3$. Else, we decrement $C_r$ by 1 and fire $t_2$. We have two cases. **Case 1:** If $C_r$ is actually zero, it runs correctly as $t_2$ would not fire. The transition $t_3$ fires and $C_r$ remains zero. In addition, $q_i$ gets the token. **Case 2:** If $C_r$ has non-zero tokens, both transitions can fire. But runs in which $t_3$ fires are “wrong” runs. We call such transitions as incorrect. The crucial point is that in runs with incorrect transitions, $S$ is not decremented whereas $C_r$ is decremented. Hence $M(S) \neq M(C_1) + M(C_2)$ in markings reached by runs with incorrect transitions.

Note that in any run of the Petri net $P$, $q_i$ and only $q_i$ gets a token when the instruction numbered $i$ is being simulated. The following lemma proves the correctness of the reduction.

\textbf{Lemma 5.7.} In every run of $P$ reaching marking $M$, $M(S) \geq M(C_1) + M(C_2)$. Furthermore, $M(S) = M(C_1) + M(C_2)$ iff there are no incorrect transitions.
If the Minsky Machine reaches instruction \( q_n \), the Petri net \( P \) gets a token in the place \( q_n \). But, if the Minsky Machine doesn’t reach \( q_n \), it is possible that the place \( q_n \) in Petri net \( P \) gets a token because of incorrect transitions. By the above lemma, to check if there had been any incorrect transitions along the run, we just check at the end (at \( q_n \)) if \( M(S) = M(C_1) + M(C_2) \). This can done using an inhibitor arc. Thus \( q_{n+1} \) gets tokens iff the Minsky Machine reaches the instruction \( q_n \). Hence reaching instruction \( q_n \) in Minsky Machine is equivalent to asking if we can cover the marking in which all places except \( q_{n+1} \) have 0 tokens and \( q_{n+1} \) has 1 token. Since checking reachability in Minsky machines is undecidable, this shows that checking coverability in Petri nets with 2 reset and an inhibitor arc is undecidable.

Note that the above proof also shows undecidability of coverability in Petri nets with 2 transfer arcs and an inhibitor arc. Additional details, the inhibitor arc construction and the extension to transfer arcs can be found in the long version of this paper [1]. Finally, the problem of coverability in Petri nets with 1 inhibitor arc and 1 reset arc is open.

**Problem 1.** Is coverability in Petri nets with 1 reset arc and 1 inhibitor arc decidable?

## 6 Adding Transfer Arcs within and without Hierarchy

We show a reduction from Petri nets with 2 (non-hierarchical) transfer arcs to HTPN preserving reachability and deadlock-freeness. Since reachability and deadlock-freeness in Petri nets with 2 transfer arcs are undecidable [9], they are undecidable in HTPN too.

**Theorem 6.1.** Reachability and deadlock-freeness are undecidable in HTPN.

**Proof.** Let \( N \) be a Petri net with 2 transfer arcs as shown in the Figure below. Here, one transfer arc is from \( p_1 \) to \( p_3 \) via \( t_1 \), while the other is from \( p_2 \) to \( p_4 \) via \( t_2 \). The transitions \( t_3 \) and \( t_4 \) are representative of other transitions to and from \( p_1 \). W.l.o.g. we assume that there is no arc from \( p_1 \) to \( t_2 \). If this is not the case, we can add a place and transition in between to create an equivalent net without adding any deadlocked reachable marking (see [1] for details). The construction of the corresponding HTPN is shown in the diagram below.

Six transfer arcs have not been shown in the construction above for clarity. These are:

- From \( p_1 \) to \( p_3 \) through \( t'_1 \).
- From \( p_1 \) to \( p'_1 \) through \( t_2 \).
- From \( p_1 \) to \( p'_1 \) through \( t'_2 \).
- From \( p'_1 \) to \( p_3 \) through \( t_1 \).
- From \( p'_1 \) to \( p_1 \) through \( t_2 \).
- From \( p'_1 \) to \( p_1 \) through \( t'_2 \).
These arcs ensure that hierarchy is respected by the transfer arcs, where \( p_1 < p'_1 < p_2 \) in the hierarchy. The dotted arc from \( p_* \) to the upper dotted box represents a pre-arc from \( p_* \) to every transition in the box. Similarly, we have an arc from every transition in the upper dotted box to \( p_* \). Dotted arcs between the lower dotted box and \( p'_* \) are interpreted similarly.

The intuitive idea behind the construction is to represent the place \( p_1 \) in the original net by two places \( p_1 \) and \( p'_1 \) in the modified net. At every marking, \( p_1 \) of the original net is represented by one of the two places \( p_1 \) or \( p'_1 \) in the modified net. Places \( p'_* \) and \( p_* \) are used to keep track of which place represents \( p_1 \) in the current marking. Everytime the transition \( t_2 \) fires, the representative place swaps. Let the original net be \( (P, T, F) \) and the constructed net be \( (P', T', F') \). The initial marking \( M'_0 \) is given by \( M'_0(p_*) = 1, M'_0(p'_*) = M'_0(p'_1) = 0 \), and \( M'_0(p) = M_0(p) \) for all other \( p \in P \). Now, given marking \( M \) of original net, we define the set \( S^{ext} = \{ A_M, B_M \} \), where,

\[
A_M(p) = \begin{cases} 
M(p) & p \notin \{ p_1, p'_1, p'_*, p_* \} \\
1 & p = p_* \\
0 & p \in \{ p'_*, p'_1 \} \\
M(p_1) & p = p_1 
\end{cases}
\]

\[
B_M(p) = \begin{cases} 
M(p) & p \notin \{ p_1, p'_1, p'_*, p_* \} \\
1 & p = p'_* \\
0 & p \in \{ p_*, p_1 \} \\
M(p_1) & p = p'_1 
\end{cases}
\]

Claim 6.1. Marking \( A_M \) or \( B_M \) is reachable from \( M'_0 \) in the constructed net iff marking \( M \) is reachable from \( M_0 \) in the original net.

The proof of the claim is given in [1]. The proof of Theorem 6.1 follows from this claim.

Corollary 6.2. Coverability is undecidable in HITPN.

Proof. From Theorem 5.6, it follows that coverability is undecidable in Petri nets with two transfer arcs and one inhibitor arc. Now, we can perform a construction similar to the one above to reduce the coverability of this net to the coverability problem of a net in HITPN.

6.1 Hardness of Termination in HITPN

Termination in HIRcTPN was shown to be decidable in Section 4.1. Termination in HTPN is also decidable, as it is known that termination in transfer Petri nets is decidable. However, termination in HITPN, which subsumes the above two problems, is as hard as the positivity problem – a long standing open problem about linear recurrent sequences ([19],[18]). We prove this by reducing the positivity problem to termination in HITPN.

Definition 6.3 (Positivity Problem). Given a matrix \( M \in \mathbb{Z}^{n \times n} \) and a vector \( v_0 \in \mathbb{Z}^n \), is \( M^k v_0 \geq 0 \) for all \( k \in \mathbb{N} \)?

Given matrix \( M \in \mathbb{Z}^{n \times n} \) and a vector \( v_0 \in \mathbb{Z}^n \), we construct a net \( N \in \text{HITPN} \) such that \( N \) does not terminate the coverability problem of a net in HITPN if \( M^k v_0 \geq 0 \) for all \( k \in \mathbb{N} \). Consider the following while loop program \( v = v0; \text{while} \ (v > 0) \ v = Mv \). Clearly, this program is non-terminating if \( M^k v_0 \geq 0 \) for all \( k \). We construct a net \( N \) which simulates this linear program. The net \( N \) contains two phases, a forward phase that has the effect of multiplying \( v \) by \( M \), and a backward phase that mimics assigning the new vector \( M \cdot v \).
on Petri Nets with Hierarchical Special Arcs

\[ M = \begin{bmatrix}
1 & -4 & 7 \\
2 & -5 & -8 \\
-3 & -6 & 9 \\
\end{bmatrix} \]

Figure 3 Forward phase (right) simulating matrix \( M \) (left).

computed in the forward phase back to \( v \). We also check for non-negativity in the backward phase, and design the net \( N \) to terminate if any component becomes negative.

**Forward Phase.** The construction of the forward phase Petri net for a general matrix is explained below. An example of the construction is shown in Figure 3. We have \( n \) places, \( u_1, u_2, \ldots, u_n \), corresponding to the \( n \) components of vector \( v \). Each place \( u_i \) is connected to a transition \( t_i \) with a pre-arc of weight 1. Each \( t_i \) also has a post-arc to a new place \( u_{ij} \) for \( 1 \leq i, j \leq n \) with a weight \( |M_{ji}| \), i.e. the absolute value of the \((j, i)\)th entry of matrix \( M \), corresponding to \( v_i \)'s contribution to the new value of \( v_j \). Finally, we have places \( u_1', u_2', \ldots, u_n' \), corresponding to the \( n \) components of the new value of vector \( v \). Each place \( u_j' \) is connected to place \( u_{ij} \) by a transition \( t_{ij} \), with both the arcs being weighted 1. If \( M_{ji} \geq 0 \), we make use of the fact that \( u_{ij} \) has a pre-arc to \( t_{ij} \) and \( t_{ij} \) has a post-arc to \( u_j' \). This has the effect of adding the value of \( u_{ij} \) to \( u_j' \). On the other hand, if \( M_{ji} < 0 \), then we make use of the fact that both \( u_{ij} \) and \( u_j' \) have pre-arcs to \( t_{ij} \) to subtract \( u_{ij} \) from \( u_j' \). The above run simulates the forward phase, in effect multiplying the vector \( v \), represented by the column of \( u_i \) in Figure 3, by \( M \), and storing the new components in the column on \( u'_i \).

To simulate the while loop program, we need to copy back each \( u'_i \) to \( u_i \), while performing the check that each \( u'_i \) is non-negative.

**Backward phase.** The copy back in the backward phase (Figure 4) is effected by a transfer arc from \( u'_i \) to \( u_i \) via transition \( t_R \). To ensure that the backward phase starts only after the forward phase completes (else, partially computed values would be copied back), we introduce a new place \( G \). The place \( G \) stores as many tokens as the total number of times each transition \( t_{ij} \) fires, and has a pre-arc weighted 1 to each transition \( t_{ij} \). The emptiness of \( G \) ensures that each \( t_{ij} \) has completed its firings in the current loop iteration. An inhibitor arc from \( G \) to \( t_R \) ensures that the forward phase completes before \( t_R \) fires. We introduce place \( G' \) which computes the initial value of \( G \) for the next loop iteration. \( G' \) has an arc
connected to $t_{ij}$ with weight $\sum_{k=1}^{n} |M_{kj}|$. If $u_j'$ has a pre-arc to $t_{ij}$, then $G'$ has a pre-arc to $t_{ij}$, while if $t_{ij}$ has a post-arc to $u_j'$, then it also has a post-arc to $G'$. Finally, there is a transfer arc from $G'$ to $G$ via $t_R$. Once the forward phase finishes, the place $G$ becomes empty. Hence, the only transition that can fire is $t_R$, which completes the backward phase in one firing. Combining the forward and backward phases, we obtain a net $N$ which simulates the while loop program. The initial marking assigns $(v_0)_i$, i.e. the $i$-th component of vector $v_0$, to place $u_i$, and $\sum_{1 \leq i \leq n} (\sum_{1 \leq j \leq n} |M_{ji}|)(v_0)_i$ tokens to $G$, while all other places are assigned 0 tokens. The following lemma (see [1] for details) relates termination of $N$ with the Positivity problem.

▶ **Lemma 6.4.** There exists a non-terminating run in $N$ iff $M^k v_0 \geq 0$ for all $k \in \mathbb{N}$.

Note that the net $N$ constructed above has only one transition with inhibitor and transfer arcs; hence $N$ is in $\text{T-HIPN}$ as well as in $\text{HITPN}$. Thus, we have,

▶ **Theorem 6.5.** Termination in $\text{HITPN}$ as well as $\text{T-HIPN}$ is as hard as positivity problem.

### 7 Conclusion

In this paper, we investigated the effect of hierarchy on Petri nets extended with not only inhibitor arcs (as classically considered), but also reset and transfer arcs. For four of the standard decision problems, we settled the decidability for almost all these extensions using different reductions and proof techniques. As future work, we are interested in questions of boundedness and place-boundedness in these extended classes. We would also like to explore further links to problems on linear recurrences. We leave open one technical question of coverability for Petri nets with 1 reset and 1 inhibitor arc (without hierarchy).

**Acknowledgments.** We thank Alain Finkel and Mohamed Faouzi Atig for interesting and insightful discussions and pointers to earlier results and reductions.
References


